Experience with FS_0 as a framework theory

Sean Matthews,
Alan Smaill,
David Basin*

Abstract

Feferman has proposed a system, FS_0, as an alternative framework for encoding logics and also for reasoning about those encodings. We have implemented a version of this framework and performed experiments that show that it is practical. Specifically, we describe a formalisation of predicate calculus and the development of an admissible rule that manipulates formulae with bound variables. This application will be of interest to researchers working with frameworks that use mechanisms based on substitution in the lambda calculus to implement variable binding and substitution in the declared logic directly. We suggest that meta-theoretic reasoning, even for a theory using bound variables, is not as difficult as is often supposed, and leads to more powerful ways of reasoning about the encoded theory.

§ 1 Introduction: why metamathematics?

A logical framework is a formal theory that is designed for the purpose of describing other formal theories in a uniform way, and for making the work of building proofs of theorems in those theories easy. Given a description T of a theory in a framework, a proof of a proposition A in that theory becomes a proof of a proposition in the framework theory that the encoding of A has the property 'provable in T'. Such a proof is a meta-theorem, and a natural question to explore is whether we can prove more general theorems in the framework theory. Furthermore, how can such metamathematics be used to increase the power and usability of theorem proving systems in practice?

§ 1.1 The deduction theorem A good example of the advantage that formal metamathematics can offer is the deduction theorem [13] for a logic presented in a Hilbert style. If the notation $\vdash_T A$ means that A is provable

*{sean,smaill,basin}@ai.ed.ac.uk

in the object-theory T, and $T[B]$ is the theory T extended with the axiom B, then the deduction theorem says

$$\text{If } \vdash_{T[B]} A, \text{ then } \vdash_T B \to A.$$

This is an interesting theorem for several reasons. First, it is an explicitly meta-level result, relating a theory to an extended version of itself; it cannot be stated for theory T inside theory T in a way so that it can be used there. Secondly, it is a perfect example of a meta-theorem that makes work in the object-theory easier — the deduction theorem is often the first theorem proven about a logic (either that, or there is a proof that it does not hold). Thirdly, proofs of theorems in the object-logic can be built in fewer steps by exploiting the meta-theorem. We *know* that the object level proof can be built if it is really needed, and that is enough; we do not usually have to build it.

But usually there is no way of sharing the sort of knowledge embodied in the deduction theorem properly with a proof development system (PDS) for T. The most that is usually possible, if the PDS has a tactic facility, is for the user to inspect the proof of the meta-theorem and build a tactic that constructs corresponding object-level proofs. (Here the meta-theorem amounts to the description and verification of an algorithm — a tactic — that takes a proof of $\vdash_{T[B]} A$ and transforms it into a proof of $\vdash_T B \to A$.) But this is unsatisfactory: the deduction theorem justifies a simple constant time syntactic transformation on an object-level goal, but a corresponding tactic will take time dependent on the size of the proof supporting that goal.

To be fair, the deduction theorem is a slightly artificial example, if only because it is such a useful result that the logic of any PDS that is usable in practice is presented in a way (sequent calculus or natural deduction) that provides it as a basic rule. However, it does illustrate very effectively the gains that meta-level reasoning can provide.

§ 1.2 Possible advantages and disadvantages The most obvious advantage of meta-theoretic extension then, is the speed up that it can bring to practical theorem proving. The gain is not always as dramatic as in the deduction theorem, but something is usually possible.

Another advantage is less obvious, but at least as important. Tactics are informal, in that they have no associated formal specifications which they can be compared against. Instead, what they are expected to do is an implicit side effect of the proof manipulations that they actually describe. For instance, it would probably not be obvious from analysing a tactic which implements the deduction theorem that the effect of running it would be the effect that the deduction theorem describes. This problem with informal tactics is analogous to the general problem of understanding unspecified code. Further, as tactics are combined together to build more powerful tactics, holding on to any intuition for how they behave becomes more difficult. To fix this, it is easy to imagine a system for verifying a tactic formally against a specification[1]. However, if a tactic has been verified against a specification, why run it at all? Why not just check that the preconditions hold, then directly perform the transformation described by the specification? This approach returns us to the sort of meta-level reasoning that is discussed above.

There are disadvantages too in extending the system's strength by proving meta-theorems rather than by verifying tactics. In the same way as tactics can be subtle, difficult pieces of code, meta-level proofs can also be subtle and difficult to construct. It can be difficult to find a formalisation for the meta-theory of a typical logical system that is comfortable to work in. In particular, there are often supposed to be problems with the formalisation of bound variables. Even without such specific problems, it usually takes longer to construct a formally verified program (which is what a meta-theorem resembles) than it does to construct the same program informally. So a meta-theorem will not always be superior to a tactic. But when a tactic is less efficient than the implementation of a meta-theorem, or tries to do something that is very subtle, or if it is used often, then there is a good case for replacing it.

A second disadvantage of using a meta-theorem is often not so important in practice: the speed up it can provide is because it removes the need actually to construct a supporting object level object-level proof, instead, simply guaranteeing that such a proof could be constructed if necessary. But there are times when this proof is needed: if the witness term of a constructive proof is to be extracted, or a transformation tactic (i.e., a tactic from proofs to proofs, rather than from sequents to sequents) is to be applied, then it is not enough simply to know that the proof exists. One possible solution to this would be to have the meta-theory be constructive (as proposed in, e.g., [2]), then the actual proof could be reconstructed by examining the proof of the meta-theorem.

In conclusion, if we are interested in provability, rather than in building object-level proofs, and if we are prepared to do the work of verifying meta-

[1] In fact, formal pre- and post- conditions could be exploited in various ways (such as combining tactics by reasoning about the conditions, as Bundy suggests in [5]).

level assertions, then the use of theorem proving in a framework theory gives us a powerful and secure way to extend our capability to obtain results for a declared theory.

§ 1.3 Overview of paper and research contributions In the remainder of this paper we will look at what is needed from a logical framework if it is to be practically useful for formulating, proving, and using meta-theorems. In §2 we motivate and present Feferman's system FS_0 [8] which we have implemented and in which we have performed a number of experiments. FS_0 is a theory of functions and classes of expressions embedded in second-order predicate calculus. The syntax and theories of formal systems are presented in FS_0 using *finitary inductive definitions*. To illustrate, we formalise a fragment of first-order predicate calculus. In §3 and §4 we provide details on the practical aspects of doing metamathematics in FS_0. In particular, we present an example of a specific meta-theorem that we have proved: a prenex form theorem for the formalised fragment of predicate calculus. We conclude by drawing lessons from our experience and suggesting directions for future work.

Our emphasis throughout is on the practical side of meta-theory. In [8], Feferman lays out FS_0 as a theoretical framework for encoding and reasoning about logics. But, as he himself points out, in the conclusion, that 'whether implementation is feasible and what its value might be can only be judged by actually trying to carry it out'. We have done just that and we think our experiments (only one of which is reported here, others are summarised in [14]) show that FS_0 in practice does provide a suitable basis for formalising and using meta-theory. As far as we know[2] ours is the first implementation and use of the system.

More generally, we believe that our research provides further support to the claim that meta-theoretic extensibility is practical and desirable. Notable examples of previous research in this area[3] that come to mind are the work of Aiello and Weyhrauch on algebraic simplification in *FOL* [1], the work of Constable and Howe on term matching, rewriting, and simplification in *Nuprl* [12, 6], and the work of Boyer and Moore on metafunctions in their theorem proving system [4]. But there has been little other experimental work in this area. One reason, we suspect, is that meta-theoretic

[2] And as far as Feferman knows [7]

[3] We are excluding research in theorem proving (some of it very notable, such as Shankar's [17] and Berardi's [3]) in which the theorems proved happen to be of a meta-mathematical nature, but are not be used as theorem proving procedures in part of a formalised meta-theory.

reasoning has a reputation for being difficult. For example, most other logical frameworks are based on type theory or some kind of higher-order logic in which variable binding and substitution are intended to be captured by lambda-abstraction and beta-reduction because it is believed to be difficult to encode 'higher-order syntax,' and operations on this syntax, explicitly. Our work indicates that this is not so. Our formalisation of predicate calculus, and the prenex normal form theorem, involved formalising notions such as free and bound variables, alpha-convertibility, substitution, and the like. These definitions were not difficult, and we needed only to restate definitions found in standard logic texts as explicit inductive definitions. The resulting admissible[19] inference rule can be applied more efficiently than an equivalent tactic, and an application of the rule is itself the first step to skolemisation and various normal forms, and, from there, to unification procedures and other mechanised proof procedures.

§ 2 Logical Frameworks for Metamathematics

Recent work in logical frameworks has focused on the use of frameworks for constructing proofs rather than formally reasoning about such proofs in the framework itself. In this paper we wish explore this second possibility, based on Feferman's proposed framework logic.

In most of the frameworks that have been suggested, e.g., *ELF* [11], *Isabelle* [15], and *Lambda-Prolog* [10], the concern has been ease of proof in the declared theory. For instance, identifying bound variables and substitution with syntax and operations in the framework theory makes substitution in declared logics easier to deal with; however, in this approach, bound variables must be understood in terms of properties of the meta-logic, leading to a less straightforward encoding. The theorems we present in this paper would have to be understood similarly — the side conditions which we explicitly formalise would have to be treated implicitly. In general, theorems that mention binding structure explicitly, e.g., those depending on the number of occurrences of named variables, are more easily treated using our approach.

Another reason for working with Feferman's theory is that it is strong enough not only to encode the syntax and proof rules of logics but also to support metatheoretic reasoning about these logics — e.g., we may prove results by induction over object-level syntax. This is another area where the above frameworks present problems. For instance, the *ELF* was designed to be weak so that faithfulness of encoding could be demonstrated more easily. Unfortunately it also means that proving general statements in the meta-theory, as opposed to proving specific statements that correspond to particular theorems in the declared theory, is difficult.

In general, an explicit facility for constructing evaluable functions in a framework theory is useful. This can then be used to exploit meta-level results directly. Again, Feferman's system lends itself to such an implementation.

§ **2.1 The theory FS_0** Feferman's system was a result of his asking the question 'what is a formal system?'. Earlier answers were supplied by Post [16] and Smullyan [18], who looked at classes of strings closed under inductive definitions as the generalised notion — in effect, the theory of recursively enumerable classes. This is a very useful approach in some ways, but the versions presented there suffer from the problem of being unusable in practice — when practice means implementing the system on a computer and using it actually to prove theorems in a declared theory. Thus it has been ignored as a possible basis for a practical framework until recently. In the last few years however, a new proposal in the same vein has been made by Feferman [8], as a competitor to type theoretic frameworks.

Feferman's theory differs from those of Post and Smullyan most importantly in its basic data structure: strings have been abandoned in favour of S-expressions[4]. He has embedded this theory in a predicate calculus, and extended it with primitive recursive functions on, and recursively enumerable classes of, S-expressions. The most important feature of the system is the ability to represent finitary inductive definitions via these recursively enumerable classes.

A more formal description of the system is as follows. The system consists of S-expressions, functions on S-expressions, functionals for combining functions, and classes. There are also sorted variables over S-expressions, functions and classes. The whole is embedded in the second order predicate calculus (with quantification over classes and functions), with equality and membership relations.

In what follows we use the following notational conventions: v, w, x, y, z are variables in FS_0 over S-expressions, f, g, h are variables in FS_0 over functions, X, Y, Z are variables in FS_0 over classes of S-expressions, U, V are meta-level variables over formulae, F is a meta-level variable over

[4]He has been able to learn from the thirty years of experience with the *Lisp* programming language that has been accumulated by programmers. *Lisp* is based on S-expressions, and is still, even after thirty years, one of the most popular languages for symbolic computing. This is a strong argument in favour of the powerful utility of such a data structure in practice.

constant-valued functions in FS_0. Also notice that universal quantification of variables that appear free in the statements of theorems or axioms is implicit.

First there is the sort of S-expressions, $\mathbf{S}$. There is only one explicitly defined S-expression, which is 0 (but, below, we show how others are created out of variables and function applications).

There are versions of the usual functions (all of type $\mathbf{F} \equiv \mathbf{S} \rightarrow \mathbf{S}$) that occur in a primitive recursive system, as follows. The comma itself is a function, for pairing, which is defined together with the projection functions π_1 and π_2, where

$$(x, y) \neq 0$$

and

$$\pi_i(x_1, x_2) = x_i.$$

These together allow arbitrary S-expressions made up of variables, 0s and other objects of sort $\mathbf{S}$ to be put together, so

$$0, \ldots, (0, (0, 0)), \ldots, ((x, 0), ((x, 0), y)), \ldots,$$

are all S-expressions (for convenience, in the rest of this paper the comma is taken as associating to the left, so that $(x, y, z) \equiv ((x, y), z)$ rather than $(x, (y, z))$). Then there is the identity ι,

$$\iota x = x,$$

the functions κ_x,

$$\kappa_x y = x,$$

and the compare function δ,

$$v = (w, x, y, z) \rightarrow \begin{cases} \delta v = y & \text{if } w = x \\ \delta v = z & \text{if } w \neq x \end{cases}$$

$$(\neg \exists w \, \exists x \, \exists y \, \exists z (v = (w, x, y, z))) \rightarrow \delta v = 0$$

(the last of these deals with the case where δ is applied to a 'not well formed' argument, i.e., one that cannot be resolved into a quadruple). Notice that, since all functions take only one argument, function application is denoted simply by juxtaposing the name of the function and the argument, though sometimes brackets will be used for clarity. (In our implementation function applications to ground S-expressions can be immediately reduced to normal forms in the natural way without having to make direct use of these rules, by an evaluation mechanism.)

These basic functions can be combined using the three second order combinators $\mathcal{C}$, $\mathcal{P}$ and $\mathcal{R}$, which are all of type $\mathbf{F} \times \mathbf{F} \to \mathbf{F}$. The first two of these are composition and pairing of functions, so that:

$$\begin{aligned} \mathcal{C}[f,g]x &= f(gx), \\ \mathcal{P}[f,g]x &= (fx, gx). \end{aligned}$$

The third is the combinator for structural recursion on S-expressions, and is more complicated. It expects to be applied to a pair (x, y), where x is a collection of parameters for the recursion, that remains constant as the function descends through the S-expression, and y is the S-expression that the recursion is upon. So the base and step cases are:

$$\begin{aligned} \mathcal{R}[f,g](a,0) &= fa, \\ \mathcal{R}[f,g](a,(b,c)) &= g(a,b,c,\mathcal{R}[f,g](a,b),\mathcal{R}[f,g](a,c)), \\ \mathcal{R}[f,g]0 &= 0. \end{aligned}$$

(As with the definition of δ, the last of these deals with case when the argument to the function is 'not well formed'.) Notice that the combinators are second order only, so they cannot be applied to themselves, e.g., $\mathcal{C}[\mathcal{C},\mathcal{R}]$ is not well defined.

Again, for convenience, the composition of two projections $\pi_i(\pi_j x)$ is written $\pi_{ij}x$. In the same way, $\mathcal{P}[f,g,h] \equiv \mathcal{P}[\mathcal{P}[f,g],h]$, and $\mathcal{C}[f,g,h] \equiv \mathcal{C}[\mathcal{C}[f,g],h]$.

At this point, what has been defined is very similar to pure Lisp restricted to primitive recursive functions: functions on S-expressions can be defined and evaluated, but not reasoned about beyond that. Next the theory is extended with classes of S-expressions ($\mathbf{C}$), a membership relation $\in$, and various class constructor operations. The first of these is the base class $\{0\}$, where

$$x \in \{0\} \leftrightarrow x = 0.$$

There are, also, the intersection operations $\cup$ and $\cap$ of type $(\mathbf{C} \times \mathbf{C} \to \mathbf{C})$ which have their obvious axioms

$$\begin{aligned} x \in X \cup Y \leftrightarrow x \in X \vee x \in Y, \\ x \in X \cap Y \leftrightarrow x \in X \wedge x \in Y, \end{aligned}$$

and the function inverse operation $\cdot^{-1}\cdot$ of type $\mathbf{F} \times \mathbf{C} \to \mathbf{C}$ where

$$x \in f^{-1}X \leftrightarrow fx \in X.$$

These operations are enough between them to provide comprehension for open formulae built from conjunctions, disjunctions and equality between terms of sort **S** in FS_0, as follows. Given such a formula U, with only one free variable over S-expressions, x, proceed as follows. While possible, replace some term of the form (fx, gx) with $\mathcal{P}[f, g]x$ or some term of the form $f(gx)$ with $\mathcal{C}[f, g]x$. Then replace each of the equalities of the form $fx = gx$ with $\mathcal{P}[f, g, \kappa_0, \kappa_{(0,0)}]x = 0$ and then all equalities of the form $fx = 0$ or $0 = fx$ with $x \in f^{-1}\{0\}$. Finally, repeatedly replace some term of the form $x \in X \vee x \in Y$ with $x \in X \cup Y$ or some term of the form $x \in X \wedge Y$ with $x \in X \cap Y$. This will result in a formula U^* of the form $x \in X$ where $\vdash_{FS_0} U^*$ iff $\vdash_{FS_0} U$.

This leaves only the constructor for recursively enumerable classes $\mathcal{I}_2$ (of type $\mathbf{C} \times \mathbf{C} \to \mathbf{C}$) to be defined. This allows us to describe in FS_0 any class Z, which can be informally defined as a base class X closed under a two place rule

$$\frac{x \in Z \qquad y \in Z}{z \in Z} U(z, x, y)$$

(where $U(z, x, y)$ is an open formula in FS_0 with free variables x, y and z). Formally, if Y is a class such that $(z, x, y) \in Y \leftrightarrow U(z, x, y)$, then $Z \equiv \mathcal{I}_2(X, Y)$ and:

$$\begin{aligned} x \in X &\to x \in Z, \\ \exists x \, \exists y (x, y \in Z \wedge (z, x, y) \in Y) &\to z \in Z. \end{aligned}$$

(which, with the induction rules described below, gives comprehension over Σ^0_1 relations, or the recursively enumerable classes). FS_0 only explicitly allows inductive definitions with two dependents, but this is all that is needed (as we explain in §2.2).

Now enough machinery has been described so that it is possible to do the same sort of object-level theorem proving as is possible in systems like *ELF*, though without the extra facilities that make *ELF* easy to use. An encoding of a language can be defined using the inductive definition mechanism of FS_0; theories can be defined in the same way.

But it is not yet possible to prove the sorts of generalised statments about defined theories which were mentioned earlier, which is our objective. To do this we need to be able to prove general statements about classes, and for this purpose there are also two induction axioms: induction over inductively defined classes, and induction over S-expressions. Induction over inductively defined classes is used to show that the defined class is a subset of some other class, i.e., (if $X \subset Y$ is an abbreviation for $\forall x(x \in X \to x \in Y)$, where x does not occur free in X or Y)

$$X \subset Z \to \forall w\, \forall x\, \forall y(x \in Z \to y \in Z \to (w, x, y) \in Y \to w \in Z) \\ \to \mathcal{I}_2(X, Y) \subset Z.$$

And induction over the S-expressions is just

$$0 \in X \to \forall x\, \forall y(x \in X \to y \in X \to (x, y) \in X) \to \forall x(x \in X).$$

§ 2.2 How $\boldsymbol{FS}_0$ is used While FS_0 classes are in principal enough to describe any formal system arising in practice, it is not obvious that this system is actually usable. To give some idea of how FS_0 does work, below is a part of the definition of a fragment of the language of first-order predicate logic.

We introduce further notational conventions at this point to allow us to discuss the declared language. We take A, B, C to vary over formulae in the language of first-order predicate logic, and a, b to vary over variables in that language, and Q to vary over quantifiers in that language. A string in double inverted commas, such as "*var*", is taken as the name of a defined constant S-expression in FS_0, and an expression of the object language in quotation marks $\ulcorner \cdot \urcorner$ stands for the encoding of that expression in FS_0.

In this encoding the components of the language are all labeled with a constant in the left hand part of the representing S-expression so that they can be distinguished. For instance, the set of variables is defined as the class of S-expressions which have the constant "*var*" (which in turn is defined as an S-expression) in the left hand part, where anything can be placed in the right hand part. So the definition can be thought of as the class

$$var \equiv \{(\text{“}var\text{”}, x) \mid x \in S\}$$

which, in the proper language of FS_0, is the class comprehending all instances y for which the formula

$$\pi_1 y = \text{“}var\text{”}$$

is provable.

Then, if the class of atomic predicates ap has been defined already, it is possible to give a definition of the class of well formed formulae in the language of predicate logic where the only propositional connectives are disjunction and negation. First, new distinct constants have to be defined for labeling the various components: "$\forall$", "$\exists$", "$\vee$" and "$\neg$". Then the rule defining the language are given as the inductive definition:

$$\begin{array}{rcl} \mathit{wff}(\ulcorner A \urcorner) & \longleftarrow & \mathit{ap}(\ulcorner A \urcorner). \\ \mathit{wff}(\ulcorner A \vee B \urcorner) & \longleftarrow & \mathit{wff}(\ulcorner A \urcorner) \wedge \mathit{wff}(\ulcorner B \urcorner). \\ \mathit{wff}(\ulcorner \neg A \urcorner) & \longleftarrow & \mathit{wff}(\ulcorner A \urcorner). \\ \mathit{wff}(\ulcorner QaA \urcorner) & \longleftarrow & \mathit{wff}(\ulcorner A \urcorner) \wedge \mathit{var}(\ulcorner a \urcorner). \end{array}$$

where the arrows indicate an inductive definition (see §2.3); so, for instance, the first line says that A is in wff if A is an atomic predicate (i.e., in ap).

This is formalised in a way suitable for the inductive definition mechanism of FS_0, as follows:

$$\begin{array}{rcl} \mathit{wff}_b & \equiv & \mathit{ap} \\ \mathit{wff}_s^1 & \equiv & \{((\text{"}\vee\text{"}, (x, y)), x, y) \mid x, y \in \mathbf{S}\} \\ \mathit{wff}_s^2 & \equiv & \{((\text{"}\neg\text{"}, x), x, y) \mid x, y \in \mathbf{S}\} \\ \mathit{wff}_s^3 & \equiv & \{((Q, (v, x)), x, y) \mid v \in \mathit{var}, Q \in \{\text{"}\exists\text{"}, \text{"}\forall\text{"}\}, x, y \in \mathbf{S}\} \end{array}$$

(where the b and s subscripts indicate base and step cases — notice that only wff_s^1 uses the y in the inductive definition; in the others it is ignored). The class of formulae is the closure under the class of atomic predicates of these definitions and can be written simply as

$$\mathit{wff} \equiv \mathcal{I}_2(\mathit{wff}_b, \mathit{wff}_s^1 \cup \mathit{wff}_s^2 \cup \mathit{wff}_s^3).$$

Then, for instance, the formula $\forall aA \vee \neg(B \vee C)$ can be encoded as the S-expression

$$(\text{"}\vee\text{"}, (\text{"}\forall\text{"}, (\ulcorner a \urcorner, \ulcorner A \urcorner)), (\text{"}\neg\text{"}, (\text{"}\vee\text{"}, (\ulcorner B \urcorner, \ulcorner C \urcorner)))).$$

However in future we usually take advantage of the notational convention defined earlier, and write this instead as $\ulcorner \forall aA \vee \neg(B \vee C) \urcorner$.

We also provide an abstract sequent calculus presentation of the logical theory. A sequent (in this case with an single consequent) is defined simply as a list of encoded formulae, so that the sequent $A_1, \ldots, A_n \vdash A$ is encoded as $(0, A_1, \ldots, A_n, A)$ (the 0 on the left marks the end of the list, and corresponds to the 'null' in Lisp).

Then a rule, which, in a sequent calculus, is of the form:

$$\frac{\Gamma_1 \vdash A_1, \ldots \Gamma_n \vdash A_n}{\Gamma \vdash A}$$

is defined as a relation between a list of sequents and a sequent. This relation in turn can be conveniently represented as a non-empty list of sequents, the head being the goal, and the tail the subgoals. So if the base class of sequents, $A \vdash A$, is defined

$$base \equiv \{(0, x, x) \mid x \in wff\}$$

then a schema for sequent calculus presentations in the language *wff* is the closure of *base* under whatever the rules are. So we write it in the schematic form

$$SC\langle RS\rangle \equiv \mathcal{I}_2(base, RS)$$

(where $SC\langle RS\rangle$ is a piece of syntactic sugar that abbreviates for the expression on the the right and RS is the set of rules for the theory). Then a presentation of first-order predicate logic using the language *wff* is a matter of defining an appropriate set of rules R^{PK} and then defining

$$PK \equiv SC\langle R^{PK}\rangle.$$

Now a proposition A is provable in first-order predicate logic iff the encoding of $\vdash A$ (the sequent with an empty context and the consequent A) is in PK. This definition of a schema for sequent calculus presentations shows why a constructor for inductive definitions with more than two dependents is not needed: the inductive definition of lists of objects requires only two dependents, and then inductive definitions dependent on arbitrary numbers of dependents can be defined in terms of inductive definitions which are dependent on a single list.

The definition of a rule can itself depend on complex inductive definitions, for instance the rule for right existential elimination depends on the definition of substitution, i.e.,

$$\frac{\Gamma \vdash A[t/a]}{\Gamma \vdash \exists a A}\exists\text{–R}$$

would be defined in FS_0 as

$$\exists\text{–R} \equiv \{(0, (x, y'), (x, (\text{“}\exists\text{”}, (z, y)))) \mid (y', y, v, z) \in sub_C\}$$

(where x, y, y', v, z correspond to $\Gamma, A, A[t/a], t, a$). And then sub is the class of four place relations between the result of a substitution, the formula to be substituted into, a term to be substituted in, and a variable to be substituted for. It is simply another inductively defined class.

$$
\begin{array}{rcl}
sub(\ulcorner QaA\urcorner, \ulcorner QaA\urcorner, \ulcorner t\urcorner, \ulcorner a\urcorner). & & \\
sub(\ulcorner t\urcorner, \ulcorner a\urcorner, \ulcorner t\urcorner, \ulcorner a\urcorner). & & \\
sub(\ulcorner a\urcorner, \ulcorner a\urcorner, \ulcorner t\urcorner, \ulcorner b\urcorner) & \longleftarrow & \ulcorner a\urcorner \neq \ulcorner b\urcorner. \\
& \vdots & \\
sub(\ulcorner QbA'\urcorner, \ulcorner QbA\urcorner, \ulcorner t\urcorner, \ulcorner a\urcorner) & \longleftarrow & \ulcorner b\urcorner \neq \ulcorner a\urcorner \wedge \ulcorner b\urcorner \textit{ not free in } \ulcorner t\urcorner \wedge \\
& & sub(\ulcorner A'\urcorner, \ulcorner A\urcorner, \ulcorner t\urcorner, \ulcorner a\urcorner). \\
sub(\ulcorner Qb'A'\urcorner, \ulcorner QbA\urcorner, \ulcorner t\urcorner, \ulcorner a\urcorner) & \longleftarrow & \ulcorner b\urcorner \neq \ulcorner a\urcorner \wedge \\
& & \ulcorner b\urcorner \textit{ free in } \ulcorner t\urcorner \wedge \\
& & \ulcorner b'\urcorner \textit{ not free in } \ulcorner t, A\urcorner \wedge \\
& & sub(\ulcorner A''\urcorner, \ulcorner A\urcorner, \ulcorner b'\urcorner, \ulcorner b\urcorner) \wedge \\
& & sub(\ulcorner A'\urcorner, \ulcorner A''\urcorner, \ulcorner t\urcorner, \ulcorner a\urcorner). \\
& \vdots &
\end{array}
$$

(notice that the definition renames variables to avoid capture — the complete definition can be found in any textbook). There is no dispute that this approach to substitution is not as convenient as what is used in other frameworks, but it is certainly usable, and it has some advantages, which we are able to exploit below.

It is easy to see that membership of ground S-expressions in sub is decidable, and we constructed a tactic to verify membership when it holds[5]. On top of this it is easy to see that a rule application is also a decidable step.

Before moving on, we define a little more syntactic sugar that hides the raw representations used in FS_0 for the sake of readability. Instead of writing the encoding of '$\Gamma \vdash A$ is provable in T' in FS_0 as $(\ulcorner\Gamma\urcorner, \ulcorner A\urcorner) \in T$, we will write, simply, $\ulcorner\Gamma \vdash A\urcorner \in T$.

§ 2.3 What we have implemented In the description above, there is little apparently in common between the system that is described as FS_0, and what is used to show FS_0 in use, though hopefully it should be clear

[5]It would be possible to provide a function F that satisfied the definition of sub so that $(F(x, y, w), x, y, w) \in sub$, which would be a one-step operation that did not need to invoke a tactic at all; such development is described in [14].

that the sugared version does not do anything that is not possible in the original syntax.

In fact, to make FS_0 usable, we have implemented a tactic based theorem prover for a sequent style presentation, inside a Prolog interpreter. We have also implemented various facilities that translate between a version of the notation used above and the 'raw' form of FS_0. So we can automatically define classes specified in the form

$$\{(x_1, \ldots, x_n) \mid U(x_n, \ldots, x_n)\}$$

(where U is a formula in FS_0, containing free variables $x_1, \ldots x_n$). And also, in the same manner, inductive definitions made up of clauses of the form

$$S(x_1, \ldots x_n) \longleftarrow U(x_1, \ldots x_i)$$

(with certain restrictions because only inductive definitions with two dependents are explicitly allowed). The access to Prolog as a tactic language means, too, that a lot of the work described in the rest of this paper has been heavily automated.

In use the system looks as follows. When the system is used for proving theorems in FS_0, the display looks like:

```
incomplete autotactic(idtac)
v6. (((0,(p2 of v1,p1 of v1),implies),sequent),provableseq
    in proofsandlistsC)
==> \/(v20
         :: ((v20,proof)in proofsandlistsC)#
            ((p2 of v20)
              = (sequent,
                  (0,
                     (implies,(conj,(pn(1,1,1,1,2)of v2,
                                     pn(1,1,2)of v1)),
                               pn(2,1,2)of v1)))))
by _
```

This shows a step in the construction of a meta-theorem. The goal states the existence of a proof object for a schematic object level goal, and the hypothesis `v6` is an induction hypothesis. The user can also see the object level schematic sequent, in the language of the declared logic, as follows.

```
[] >> (*object*(pn(1,1,1,1,2)of v2) &
       *object*(pn(1,1,2)of v1)) =>
          *object*(pn(2,1,2)of v1)
```

which, in terms of the notation used in this paper, is the goal

$$\ulcorner \vdash (\pi_{11112}x \wedge \pi_{112}y) \rightarrow \pi_{212}y\urcorner.$$

Currently the system is experimental, but demonstrates the feasibility of these ideas. With additional work on the interface, it could form the basis for a practical system.

§ 3 How to do metamathematics in FS_0

From the description given above it should be clear that FS_0 theoretically offers the capability to develop the sort of metamathematical extensions that are discussed in the initial part of this paper. We now discuss more practical details.

§ 3.1 Formalising meta-theoretic extensions When we extend a declared logic with an admissible inference rule, we want the rule to function as a primitive rule of the original theory. That is, it should have the form

$$\frac{\Gamma' \vdash A'}{\Gamma \vdash A} Rule$$

(in general, there may be several premisses). This may be expressed within FS_0 as

$$\begin{aligned} &DR\langle AR\rangle \equiv \\ &\qquad (0, \ulcorner\Gamma' \vdash A'\urcorner, \ulcorner\Gamma \vdash A\urcorner) \in AR \rightarrow \ulcorner\Gamma' \vdash A'\urcorner \in PK \rightarrow \ulcorner\Gamma \vdash A\urcorner \in PK. \end{aligned}$$

Here AR is a schematic variable ranging over relations between lists of sequents — in this case, a singleton list — and a sequent.

However, in using this formulation to simulate an inference rule, we need to supply both $\ulcorner\Gamma' \vdash A'\urcorner$, and a proof that the antecedent of the schema holds. Thus we have not yet achieved our goal of providing a new one-step rule of inference.

One approach is to write a tactic that will carry out both of these tasks, i.e., construct the new sequent, and generate a proof of the antecedent. However, this is still not genuinely one-step, as the tactic may construct arbitrarily large object-level derivations — indeed we have no guarantee that the tactic will succeed.

The solution in FS_0 is to provide a function F in FS_0 so that

$$\{(0, Fx, x) \mid x \in \mathit{wff}\} \subset AR$$

upon which $DR\langle AR\rangle$ is enough to show that $Fx \in T \to x \in T$.

The definition of F is not usually immediate. If it were, we could define AR directly as the class

$$AR \equiv \{(Fx, x) \mid x \in \mathit{wff}\}.$$

In practice, it is easier to work with an intermediate relational definition, which allows us to separate what we want to do from how we will go about doing it. This is particularly convenient in FS_0 since it is designed to be good at working with recursively enumerable classes, not recursive functions, and its function construction and reasoning facilities are (we have found in practice) trickier, and more frustrating, to use than those for recursively enumerable classes — not least because they are based on primitive recursion. Moreover, the specification of AR will usually translate naturally into a recursively enumerable class. The same, translated into a computationally efficient primitive recursive function, may not be so comprehensible.

§ 4 Doing metamathematics in $\boldsymbol{FS}_0$

Now we present a particular example of what has been described above in general form; the example we have chosen is the definition and construction of a rule for quantifier prenex form for predicate logic. The rule is provided for the theory PK defined above, where the only connectives are the quantifiers, disjunction and negation. The prenex form theorem says:

> For any formula A in classical first-order predicate logic PK there exists another formula $A^* \equiv Qa_1 \ldots Qa_n B$ where each a_i is distinct from the others, each Q is a universal or existential quantifier, B is open, and such that $\vdash A \leftrightarrow A^*$. From this, the admissible rule $\Gamma \vdash A$ iff $\Gamma \vdash A^*$ follows.

§ 4.1 Defining the relation Below we define a relation between A and a corresponding prenex form. The relation we formalise is designed to suggest a subsequently defined function that computes normal forms. The definition here deals only with a fragment of the language of PK, so it is not possible to use DeMorgan's laws for instance. The details are as follows:

$$\begin{aligned}
\mathit{pf}(\ulcorner A\urcorner, \ulcorner A\urcorner) &\longleftarrow \mathit{ap}(\ulcorner A\urcorner).\\
\mathit{pf}(\ulcorner QaA\urcorner, \ulcorner QaA^*\urcorner) &\longleftarrow \mathit{pf}(\ulcorner A\urcorner, \ulcorner A^*\urcorner).\\
\mathit{pf}(\ulcorner \neg A\urcorner, \ulcorner B\urcorner) &\longleftarrow \mathit{pf}(\ulcorner A\urcorner, \ulcorner A^*\urcorner) \wedge \mathit{pni}(\ulcorner A^*\urcorner, \ulcorner B\urcorner).\\
\mathit{pf}(\ulcorner A \vee B\urcorner, \ulcorner C\urcorner) &\longleftarrow \mathit{pf}(\ulcorner A\urcorner, \ulcorner A^*\urcorner) \wedge \mathit{pf}(\ulcorner B\urcorner, \ulcorner B^*\urcorner) \wedge
\end{aligned}$$

$$joinor(\ulcorner A^* \urcorner, \ulcorner B^* \urcorner, \ulcorner C \urcorner).$$

where the subsidiary definitions are:

$$\begin{array}{rcl} pni(\ulcorner A \urcorner, \ulcorner \neg A \urcorner) & \longleftarrow & \ulcorner A \urcorner \neq \ulcorner QaC \urcorner. \\ pni(\ulcorner QaA \urcorner, \ulcorner \overline{Q} a A^* \urcorner) & \longleftarrow & pni(\ulcorner A \urcorner, \ulcorner A^* \urcorner). \end{array}$$

and

$$\begin{array}{rcl} joinor(\ulcorner A \urcorner, \ulcorner B \urcorner, \ulcorner A \vee B \urcorner) & \longleftarrow & \ulcorner A \urcorner \neq \ulcorner QaC \urcorner \wedge \ulcorner B \urcorner \neq \ulcorner QaC \urcorner. \\ joinor(\ulcorner QaA \urcorner, \ulcorner B \urcorner, \ulcorner Qa^* C \urcorner) & \longleftarrow & joinor(\ulcorner A[a^*/a] \urcorner, \ulcorner B \urcorner, \ulcorner C \urcorner) \\ & & \wedge \ulcorner a^* \urcorner \notin \ulcorner A \urcorner, \ulcorner B \urcorner. \\ joinor(\ulcorner B \urcorner, \ulcorner A \urcorner, \ulcorner C \urcorner) & \longleftarrow & joinor(\ulcorner A \urcorner, \ulcorner B \urcorner, \ulcorner C \urcorner). \end{array}$$

(Here the notation $\overline{Q}$ just means 'the other quantifier than Q', e.g., $\overline{\forall} \equiv \exists$, and $\ulcorner a \urcorner \notin \ulcorner A \urcorner$ means that a does not occur free in A; *pf* stands for 'predicate form' and *pni* for 'push negation in'). These definitions can easily be converted into the definitions of recursively enumerable sets in FS_0, in the same way as *wff* earlier; so (like with *wff*) classes of the form $pf_b, pf_s^1, \ldots, joinor_b, \ldots$ etc. will actually be built by the system.

§ 4.2 Verifying the relation *pf* Since this is supposed to correspond to an admissible rule, we will want to prove an instance of $DR\langle\cdot\rangle$, but first we show that

$$\vdash_{FS_0} (\ulcorner A \urcorner, \ulcorner A^* \urcorner) \in pf \rightarrow \ulcorner \vdash A \leftrightarrow A^* \urcorner \in PK \qquad (*)$$

where the usual syntactic abbreviations for the various connectives hold, i.e.,

$$\begin{array}{rcl} A \leftrightarrow B & \equiv & (A \rightarrow B) \wedge (B \rightarrow A), \\ A \wedge B & \equiv & \neg(\neg A \vee \neg B), \\ A \rightarrow B & \equiv & \neg A \vee B. \end{array}$$

§ 4.2.1 An appropriate form of the goal Since *pf* has been defined by inductive definition, the obvious way to prove $(*)$ is by induction over classes, but this is not yet possible; the goal has first to be rewritten to

$$pf \subset \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1} PK \qquad (*')$$

with the definitions:

$$\begin{array}{rcl}
\mathit{iff}_F &=& \mathcal{C}[\mathit{and}_F, \mathcal{P}[\mathit{imp}_F, \mathcal{C}[\mathit{imp}_F, \mathit{rev}_F]]], \\
\mathit{and}_F &=& \mathcal{C}[\mathit{neg}_F, \mathit{or}_F, \mathcal{P}[\mathcal{C}[\mathit{neg}_F, \pi_1], \mathcal{C}[\mathit{neg}_F, \pi_2]]], \\
\mathit{imp}_F &=& \mathcal{C}[\mathit{or}_F, \mathcal{P}[\mathcal{C}[\mathit{neg}_F, \pi_1], \pi_2]], \\
\mathit{or}_F &=& \mathcal{P}[\kappa_{``\vee"}, \iota], \\
\mathit{neg}_F &=& \mathcal{P}[\kappa_{``\neg"}, \iota], \\
\mathit{rev}_F &=& \mathcal{P}[\pi_2, \pi_1].
\end{array}$$

Then it is fairly easy to see, by simplifying the function applications, that $(*)$ if $(*')$. I.e.,

$$\mathit{pf} \subset \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1} PK \leftrightarrow \forall x (x \in \mathit{pf} \to x \in \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1} PK)$$

and

$$x \in \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1} PK \leftrightarrow \mathcal{P}[\kappa_0, \mathit{iff}_F] x \in PK$$

and, from the definitions above,

$$\begin{array}{l}
\mathcal{P}[\kappa_0, \mathit{iff}_F] x \\
\quad = \quad (0, \mathit{iff}_F x) \\
\quad = \quad (0, \mathit{and}_F(\mathit{imp}_F x, \mathit{imp}_F(\mathit{rev}_F x))) \\
\quad = \quad (0, \mathit{and}_F(\mathit{or}_F((\mathit{neg}_F(\pi_1 x), \pi_2 x), (\mathit{neg}_F(\pi_2 x), \pi_1 x)))) \\
\quad = \quad (0, \mathit{and}_F(\ulcorner \neg \pi_1 x \vee \pi_2 x \urcorner, \ulcorner \neg \pi_2 x \vee \pi_1 x \urcorner)) \\
\quad = \quad (0, \mathit{and}_F(\ulcorner \pi_1 x \to \pi_2 x \urcorner, \ulcorner \pi_2 x \to \pi_1 x \urcorner)) \\
\quad \vdots \\
\quad = \quad \ulcorner \vdash \pi_1 x \leftrightarrow \pi_2 x \urcorner.
\end{array}$$

Notice that there is a slight abuse here of the conventions we have defined: terms of FS_0 such as variables and their projections appear as part of the notation for encoded sequents of the declared theory. In fact x is an FS_0 variable ranging over pairs of formulae in the declared language. This is how schematic formulae are represented in our system. Fortunately, this reduction does not need to be done by hand, since the machine is able to take care of the messy details.

Then applying induction over the classes defined in §§ 4.1 to $(*')$ generates two subgoals:

the base case

$$\vdash_{FS_0} x \in pf_b \to x \in \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1} PK$$

and the step case

$$\begin{aligned}\vdash_{FS_0} x \in \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1} PK \to &\\ y \in \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1} PK \to &\\ (w, x, y) \in pf_s \to &\\ w \in \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1} PK.&\end{aligned}$$

§ **4.2.2 The base case** The base case is easy; the definition of pf_b given above is translated automatically into a formula equivalent to $x \in pf_b \leftrightarrow (\pi_1 x = \pi_2 x \wedge \pi_1 x \in ap)$, so the base case simplifies to

$$\pi_1 x \in ap \vdash_{FS_0} \ulcorner \pi_1 x \leftrightarrow \pi_1 x \urcorner \in PK$$

which follows trivially.

§ **4.2.3 The step case** The step case is, naturally, more difficult: here there are three sub-cases, depending on whether the outermost connective is a quantifier, a disjunction or a negation. Assuming that the step case has been reduced to a subgoal for each of these cases, we will work through the first (for the case where the quantifier is "$\forall$"), since it is the easiest and makes all the important points without obscuring details. The goal that has to be proven then, reduces by thinning, simplification, and rules available in R^{PK} to:

$$\ulcorner \pi_1 x \vdash \pi_2 x \urcorner, \ulcorner \pi_2 x \vdash \pi_1 x \urcorner \in PK,\ z \in var \vdash_{FS_0} \ulcorner \forall z(\pi_1 x) \vdash \forall z(\pi_2 x) \urcorner \in PK.$$

The goal here is very similar in form to a goal in the declared theory, as described earlier in this paper, except that it is schematic; x varies over pairs of formulae in the object language, and z over variables in the object language. This does not stop us treating it like an ordinary object-level goal though, so long as we realise that some of the housekeeping that is automatically checked when the goal is ground will instead have to be dealt with separately. The proof follows by eliminating the quantifier on the right using an instance of z. This can be done because z does not appear free on the hypothesis list. Then, by the fact that $\pi_2 x[z/z] = \pi_2 x$, the consequent is reduced to $\pi_2 x$. Then eliminating the quantifier of the formula on the left in the same way reduces the goal to one of the hypotheses, and the result follows.

What is most remarkable about this proof is how much of it can be done in much the same way as an ordinary object-level proof. The important difference is that some of the theory of the binding mechanism has to be developed, since some of the side conditions which, for ground terms, are decidable propositions, have now to be proven by appeal to lemmas (e.g., that fact that identity substitution is an identity operation). In fact, as long as some care is taken in their construction, FS_0 tactics designed for encoded object level proofs for PK are usable for more general reasoning, e.g., schematic PK formulae, as in this example.

§ **4.3 The new derived rule** Given the proof of $(*)$ we can now prove the appropriate instance of the $DR\langle\cdot\rangle$ schema. First it is necessary to construct an appropriate relation between a list of sequents and a sequent, which can be easily defined as

$$Pf(\ulcorner \Gamma \vdash A^* \urcorner, \ulcorner \Gamma \vdash A \urcorner) \quad \longleftarrow \quad pf(\ulcorner A \urcorner, \ulcorner A^* \urcorner),$$

(where Pf stands simply for 'sequent predicate form') then $\vdash_{FS_0} DR\langle Pf\rangle$ quickly reduces, after simplification, and cutting in $(*)$, to:

$$\ulcorner \Gamma \vdash A \urcorner \in PK, \ulcorner \vdash A \leftrightarrow A^* \urcorner \in PK \vdash_{FS_0} \ulcorner \Gamma \vdash A^* \urcorner \in PK$$

which is easily proved by a tactic for PK proofs.

Now we have achieved the first part of what is described in §3.1. So we could build a tactic that would take any $\ulcorner \Gamma \vdash A \urcorner \in PK$, construct an appropriate $\ulcorner \Gamma \vdash A^* \urcorner$, and then construct a proof disposing of the resultant subgoal

$$(\ulcorner \Gamma \vdash A^* \urcorner, \ulcorner \Gamma \vdash A \urcorner) \in Pf.$$

§ **4.4 Constructing and verifying the function** But if we stop here, we still need, each time, to prove the subgoal; we still do not have a one-step derivation at the user interface. To fix this, the next part of the development is the construction of a function pf_F that can be verified to perform the transformation automatically, inside FS_0.

The definition of pf itself corresponds to an algorithm, but the algorithm is not encodable directly in FS_0: the only computation facility available directly in FS_0 is primitive recursion on S-expressions. We have implemented a facility that is able to solve specifications for functions that use course of values recursion on S-expressions, and is easily able to deal with functions for pf and pni defined as follows:

$$\begin{array}{rcl}
pf_F(\ulcorner A \vee B \urcorner) & = & joinor_F(pf_F(\ulcorner A \urcorner), pf_F(\ulcorner B \urcorner)), \\
pf_F(\ulcorner \neg A \urcorner) & = & pni_F(pf_F(\ulcorner A \urcorner)), \\
pf_F(\ulcorner Qa A \urcorner) & = & \ulcorner Qa(pf_F(\ulcorner A \urcorner))\urcorner, \\
pf_F(\ulcorner A \urcorner) & = & \ulcorner A \urcorner, \\
pni_F(\ulcorner Qa A \urcorner) & = & \ulcorner \overline{Q} a(pni_F(\ulcorner A \urcorner))\urcorner, \\
pni_F(\ulcorner A \vee B \urcorner) & = & \ulcorner \neg(A \vee B)\urcorner, \\
pni_F(\ulcorner \neg A \urcorner) & = & \ulcorner A \urcorner, \\
pni_F(\ulcorner A \urcorner) & = & \ulcorner \neg A \urcorner.
\end{array}$$

(It should be noted that the branches are assumed to be tested in order, so that the last branch of, for instance, pf_F above, is the 'otherwise' case — it is assumed implicitly that none of the possibilities above it in the list hold). The definition of $joinor_F$ presents greater difficulties though, since its definition is not course of values. But the definition can be rewritten using course of values recursion if a secondary function is used.

$$\begin{array}{rcl}
joinor_F(\ulcorner Qa A \urcorner, \ulcorner B \urcorner) & = & \ulcorner Qa^*(joinor_F(Sub(\ulcorner a^* \urcorner, \ulcorner a \urcorner, \ulcorner A \urcorner), \ulcorner B \urcorner))\urcorner \\
 & & \text{where } \ulcorner a^* \urcorner = nfi_F(\ulcorner A, B \urcorner) \\
joinor_F(\ulcorner A \urcorner, \ulcorner B \urcorner) & = & joinor'_F(\ulcorner A \urcorner, \ulcorner B \urcorner) \\
joinor'_F(\ulcorner A \urcorner, \ulcorner Qa B \urcorner) & = & \ulcorner Qa^*(joinor'_F(\ulcorner A \urcorner, Sub(\ulcorner a^* \urcorner, \ulcorner a \urcorner, \ulcorner B \urcorner)))\urcorner \\
 & & \text{where } \ulcorner a^* \urcorner = nfi_F(\ulcorner A, B \urcorner) \\
joinor'_F(\ulcorner A \urcorner, \ulcorner B \urcorner) & = & \ulcorner A \vee B \urcorner
\end{array}$$

We now show how to prove that the relation

$$pfF \equiv \{(x, y) \mid x \in wff, y = pf_F x\}$$

is a subrelation of pf, i.e., that

$$pfF \subset pf. \tag{$**$}$$

This cannot be done in the same way as in the proof of the apparently similar $(*')$ because pfF is not given as an inductively defined class. Since we are trying to verify a function, we want to use induction over S-expressions, which we can do, indirectly, as follows: *pfF* is defined as a primitive recursive class, and therefore its complement $\overline{pfF}$ is definable in FS_0 as

$$\overline{pfF} \equiv \{(x, y) \mid y \neq pf_F x\}.$$

So, since $(**)$ is a consequence of

$$\begin{aligned} x \notin pfF \vee x \in pf & \\ \leftrightarrow \quad & x \in \overline{pfF} \vee x \in pf \\ \leftrightarrow \quad & x \in \overline{pfF} \cup pf \end{aligned}$$

and the last of these can be tackled easily by a course of values induction on S-expressions (the path of the proof simply follows the structure of the definition of the relation), the result follows.

Then, if *PfF* is defined as

$$PfF(\ulcorner \Gamma \vdash A^* \urcorner, \ulcorner \Gamma \vdash A \urcorner) \quad \longleftarrow \quad pfF(\ulcorner A \urcorner, \ulcorner A^* \urcorner),$$

it is easy to show that $DR\langle PfF \rangle$ implies

$$\ulcorner \Gamma \vdash A^* \urcorner \in PK \rightarrow \ulcorner \Gamma \vdash A \urcorner \in PK$$

where $\ulcorner A^* \urcorner$ is $pf_F(\ulcorner A \urcorner)$. This is a genuine one-step deduction, since the normalisation of $pf(A)$ can be done by the PDS itself.

At this stage we have constructed a function that transforms a formula in the language of *wff* into its prenex form, in one step, inside FS_0. Furthermore, we have proved formally, in FS_0, that this transformation is admissible in the theory *PK*, i.e., that

$$\ulcorner \Gamma \vdash A^* \urcorner \in PK \rightarrow \ulcorner \Gamma \vdash A \urcorner \in PK$$

is valid for *PK*, which is what we wanted to do.

§ 5 Conclusions and further work

We have described work showing that Feferman's proposed system FS_0 is a practical option for a framework theory, and further that it is possible to use FS_0 to prove meta-level theorems that can be used to extend a declared logic safely and efficiently. Specifically, we have shown that it is not difficult to reason about formal theories that use bound variables, and to derive admissible rules, such as the prenex example.

The total specifications for the language and theory used in the example described here consisted of about a thousand lines of definitions, but a lot of this was basic definitions and the substitution mechanism. It was also the first large definition that has been constructed with the current support facilities. A reconstruction of the same work would, we think, be smaller, and would also be more modular — as the specification is at the moment it is difficult to see which definitions are specific to the definition of *PK* and which are more general. The specification of (a slightly more sophisticated form of) the normal form theorem consisted of about a hundred lines of code, and the two parts of the proof (showing that the relation is valid, and constructing a function that satisfies the relation) took about a day to construct each.

§ **5.1 The implementation** One obvious question to ask is 'how important is the fact that we have implemented our system directly in *Prolog*?'; i.e., would FS_0 be a usable system if it was instead implemented using another framework such as *Isabelle* or the *ELF*? We would argue that such an implementation would probably not be very useful for the following reason: FS_0 is itself intended as a framework theory, so if it were implemented on top of another system then any work done in a logic declared in it would have to work through two levels of encoding to get to the machine level. Further, special care was taken to provide an efficient evaluation mechanism for function applications, and this might not be available in other approaches.

§ **5.2 Further work, and other considerations** One distinctive feature of FS_0 is that its proof-theoretic strength is very low (identical to *PRA* — primitive recursive arithmetic). This restricts the class of theorems that can be proven (the problem is particularly significant for meta-theory, which does sometimes need to appeal to stronger forms of induction than is available in *PRA*). The reason that Feferman gives for this restriction is '... to have a formulation of Godel's second incompleteness theorem which applies to systems containing PRA in one way or another'[7], which would be a challenging further piece of work. This would in turn introduce the possibility of using reflection principles to increase the proof strength of the system. Indeed, the idea that reflection be exploited in FS_0 is implicit, if not explicit, in another strand of his work, for instance in his recent paper [9].

Less ambitiously, it is clear from what is described above that the ability to define functions is important when working in FS_0. In principle, the functions that the system allows to be defined are enough to allow anything that is feasibly computable to be computed. However, using only the facilities that are directly available, defining an appropriate function can be very difficult in practice. The development of (conservative) extensions to the function definition facilities, to make this easier would be a very worthwhile step.

Another direction of research that seems to offer possibilities not so easily available in other frameworks is the notion of schematic presentations of theories. An examination of the formal proof of the theorem presented in this paper shows that, for instance, wherever the theory $PK \equiv SC\langle R^{PK}\rangle$ is used, it would be easily possible instead to prove a version of the theorem abstracted over a class EX, that used the theory $PK' \equiv SC\langle R^{PK} \cup EX\rangle$, thus proving that the rule is admissible in any extension of PK. This might offer very general facilities for sharing meta-theoretic results; but we have barely started to consider the implications of this.

References

[1] Aiello, L. and Weyhrauch, R. (1980). Using meta-theoretic reasoning to do algebra, in *5th International Conference On Automated Deduction*, (Les Arcs, France).

[2] Basin, D. and Constable, R. (1991). Metalogical frameworks, in *Proceedings of the Second Workshop on Logical Frameworks*, (Edinburgh, Scotland).

[3] Berardi, S. (1991). Girard normalization proof in LEGO, in *Proceedings of the Second Workshop on Logical Frameworks*, (Edinburgh, Scotland).

[4] Boyer, R. and Moore, J. (1981). Metafunctions: proving them correct and using them efficiently as new proof procedures, in *The Correctness Problem in Computer Science*, pages 103–184, (Academic Press).

[5] Bundy, A. (1988). The use of explicit plans to guide inductive proofs, in *9th International Conference On Automated Deduction*, pages 111–120 (Argonne, Illinois).

[6] Constable, R. and Howe, D. (1990). Implementing metamathematics as an approach to automatic theorem proving, in *A source book of formal approaches to A.I.*, ed. R.B. Banerji (North Holland, Amsterdam).

[7] Feferman, S. (1991). Private Communication.

[8] Feferman, S (1989). Finitary inductively presented logics, in *Logic Colloquim '88* (North-Holland, Amsterdam).

[9] Feferman, S. (1992). Reflecting on incompleteness, *Journal of Symbolic Logic*, **56**, 1–49.

[10] Felty, A. and Miller, D. (1988). Specifying theorem provers in a higher-order logic programming language, in *9th International Conference On Automated Deduction*, (Argonne, Illinois).

[11] Harper, H. and Honsell, F. and Plotkin, G. (1987). A framework for defining logics, in *The Second Annual Symposium on Logic in Computer Science*, pages 193–204 (IEEE).

[12] Howe, D. (1988) Computational metatheory in Nuprl, in *9th International Conference On Automated Deduction*, pages 238–257 (Argonne, Illinois).

[13] Kleene, S. (1952). *Introduction to Metamathematics* (Van Nostrand, New York).

[14] Matthews, S. (1992). Meta Level and Reflexive Extension in Mechanical Theorem Proving, University of Edinburgh, Ph.D. thesis (to appear).

[15] Paulson, L. (1986). Natural deduction proof as higher-order resolution, *Journal of Logic Programming*, **3**, 237–258.

[16] Post, E. (1943). Formal reductions of the general combinatorial decision problem, *Amer. J. Math*, **65**, 197–214.

[17] Shankar, N. (1985). Towards mechanical metamathematics. *Journal of Automated Reasoning*, **1**, 407–434.

[18] Smullyan, R. (1961). *Theory of Formal Systems* (Princeton University Press, New York).

[19] Troelstra, A.S. (1973). *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis* (Springer-Verlag, New York).