A Theory and its Metatheory in FS_0

Seán Matthews
Max-Planck-Institut für Informatik*

Abstract. Feferman has proposed FS_0, a theory of finitary inductive systems, as a framework theory that allows a user to reason both in and about an encoded theory. I look here at how practical FS_0 really is. To this end I formalise a sequent calculus presentation of classical propositional logic, and show this can be used for work in both the theory and the metatheory. the latter is illustrated with a discussion of a proof of Gentzen's Hauptsatz.

Contents

*Im Stadtwald, D-66123 Saarbrücken, Germany. <sean@mpi-sb.mpg.de>

§ 1 Introduction

§ 1.1 Background In order to be easier to use, proof development systems provide a range of facilities such as 'lemmas', tactic languages and uniform proof procedures, so that the work of building proofs can be automated as much as possible. For the most part, in fact, formal derivations of 'real' theorems can be built only because of such facilities, since they are so big that it is not practical for the user of a system to enter every atomic step by hand. In fact a lot of proofs are in essence not very big; most of the size of a formal derivation represents the work of fitting the original intuition into the given theory. A tactic language, or other similar facility, is a way for the user to automate some of this work, leaving him free to concentrate on the important parts of proofs, and also, incidentally, to speed things up, since a machine can usually build derivations much more quickly and reliably than a user manually entering rules.

Mathematics is not done with a proof development system in quite the same way as it is done in a textbook, even when the two look like one another. For instance in a book on algebra one might read

> 'If A is an abelian group, then, for all a, b in A, the equivalence
>
> $$\overbrace{(a \circ b) \circ \cdots \circ (a \circ b)}^{n \text{ times}} = \overbrace{(a \circ \cdots \circ a)}^{n \text{ times}} \circ \overbrace{(b \circ \cdots \circ b)}^{n \text{ times}},$$
>
> holds'

followed with a proof. This allows another proof, further on, to go, in one step,

$$\begin{aligned} a &= (b \circ c) \circ (b \circ c) \circ (b \circ c) \\ &= (b \circ b \circ b) \circ (c \circ c \circ c), \end{aligned}$$

with the step justified by an appeal to the theorem proved earlier. On the other hand, instead of a book, imagine a proof development system for algebra; there the theorem cannot be stated, since it is not a theorem of abelian group theory, it is, rather, a meta-theorem, a theorem *about* abelian group theory.

Even if it is not possible to state the theorem though, it is still possible, in a sense, to state the proof. Since this is a description of how to effect the transformation inside the theory, it can be translated into the tactic language as a program that does that transformation. Then it is possible to build proofs in the system almost like before; i.e.

$$\begin{aligned} a &= (b \circ c) \circ (b \circ c) \circ (b \circ c) \\ &\vdots \\ &= (b \circ b \circ b) \circ (c \circ c \circ c) \end{aligned}$$

where the gap is filled automatically by running the program. This looks the same as in the book, but there are important differences. First, there is no guarantee that the program will succeed in filling the gap, only the assurance of whoever wrote it. Secondly, even if the program succeeds, the safety of the step it allows is guaranteed by the fact that it actually builds the bridging proof, and this can take time — imagine if n is 65535.

Of course, even on a machine it is not always necessary to use a tactic; a lot of theorems can be stated in the language of group theory itself. For instance, one can say that

$$\forall a, b [\overbrace{(a \circ b) \circ \cdots \circ (a \circ b)}^{65535 \text{ times}} = \overbrace{(a \circ \cdots \circ a)}^{65535 \text{ times}} \circ \overbrace{(b \circ \cdots \circ b)}^{65535 \text{ times}}],$$

prove it, and store it as a lemma, then whenever an instance of this special fact is needed, the stored lemma can be recalled and instantiated with the appropriate a and b; there is no need to rebuild the proof. What is missing from most proof development systems is a facility like this, but at the 'meta-level', so that the general case can be stated, proved, and (re)used in the same way.

Proof development systems based on framework theories offer this possibility. Such systems are advertised as not specialised for proving theorems in some particular theory (such as group theory) but instead easily adapted to different sorts of theories (first or higher order, classical, intuitionistic or linear, modal, etc.). Another way to look at them, however, is as being specialised for proving theorems in some *framework* theory designed for the express purpose of describing other theories. The idea being that, given a formal description of a theory written in the language of the framework, it is possible to build derivations in the theory via the framework. In fact the description of the theory in the framework is a metatheory, and theorems derived in a theory declared in the framework are really simple metatheorems in the metatheory. But if simple metatheorems can be proved, why not more complex ones, e.g. the more general sorts proposed above?

§ **1.2 Outline of paper** Not all the framework theories that have been proposed are very good for proving these 'real' metatheorems — they may have been designed with other ends in mind. However Feferman has proposed as a framework the theory FS_0, which he has designed, among other things, expressly as a tool not only for formalising, and working in theories, but also

for metatheoretic analysis of them; in other words, to do exactly the sort of things I have just described.

In this paper I look at how well FS_0 behaves, in a worked example, in its designated rôles. The paper divides into three parts. The first part, sections § 2 to § 5, is structured as follows: section § 2 is background detail about FS_0 and my notation, section § 3 describes the theory that I formalise, a sequent calculus for a complete fragment of classical propositional logic, section § 4 looks at a formalisation of this theory, and section § 5 looks briefly at how easy it is to build derivations in that formalisation. The second part, sections § 6 and § 7 is a discussion of a proof in FS_0 of a fundamental metatheorem, Gentzen's Hauptsatz (also known as the cut elimination theorem), for the theory just formalised, and some possible applications. Finally the third part, sections § 8 and § 9, looks at related and possible further work, and draws some conclusions.

§ 2 The theory FS_0 and notational conventions

I refer to Feferman's paper [9] for details of FS_0. Here I will just give a quick survey, together with a description of how my notation differs from what is described there.

§ 2.1 What is FS_0 The theory FS_0 is a conservative extension of primitive recursive arithmetic, a weak second order theory of s-expressions and primitive recursive functions. It is like a version of *Pure Lisp* [15] where only certain functions can be defined, but that is supplemented with facilities for defining recursively enumerable classes, and with induction over such classes.

S-expressions are defined inductively as follows: O is an s-expression, and if a and b are s-expressions, then so is (a, b) — one can think of the comma as a function of arity two, the equivalent of `cons` in *Lisp*. For the sake of clarity, like Feferman, I take the comma as associating to the left, so that $(a, b, c) \equiv ((a, b), c)$. An important difference between FS_0 and *Lisp* is that in FS_0 the value O is used to stand for 'true', and anything else is 'false'; while in *Lisp* the situation is the other way about. Thus I can define abbreviations for particular s-expressions as

$$\begin{aligned} \mathit{True} &\triangleq O \\ \mathit{False} &\triangleq (O, O) \end{aligned}$$

(I use $\triangleq$ to indicate a declaration of formal definitional equivalence and I write defined names with an initial capital letter and variables with an initial lower-case letter — the occasional exception to this is when I use single upper-case letters as class variables, but this use will always be clear in context).

A set of simple functions for operating on s-expressions is available including, for instance, projection functions π_1 and π_2, where $\pi_1(a, b) = a$ and $\pi_2(a, b) = b$, corresponding to `car` and `cdr` of *Lisp*. There are also second order combinators for functional composition, pairing and structural recursion. However I will not, for the most part, make explicit use of these — defining functions with them is a tedious exercise and my implementation of the system has a compiler available that can take sets of conditional equations and automatically build appropriate functions out of primitive components (assuming, of course, that it

can see that the equations define a primitive recursive function). Instead I will use a notation similar to that of my implementation for the functions I define here. In that notation I can define an *And* function simply by saying that it is a solution to the equations

$$\begin{aligned} And(True, a) &= a \\ And(False, a) &= False \end{aligned}$$

(for convenience sake, I assume that equations are read in the given order, and that the conditions on an equation implicitly include the negations of the conditions on all the previous equations — also notice that the definition is not typed, so there is no requirement, for instance, that a evaluate to something that looks like the boolean constants defined above).

In FS_0 comprehension is available for Σ^0_1-formulae, and in the same way that my implementation is able to build functions automatically from equational definitions, it is also able to build sets corresponding to the comprehension of such formulae. So I can write the definition of a class A, the extension of some predicate, as

$$A \triangleq \{ (a, b, c) \mid \exists d, e, \ldots P(a, b, c, d, e, \ldots) \}$$

(where $P(a, b, c, d, e, \ldots)$ is some quantifier free formula with no free variables other than $a, b, c, d, e, \ldots$), and the system will build the concrete definition itself.

Finally, I can define a class C, that is the closure of a class B under a rule

$$\frac{b \in C \quad c \in C}{a \in C} \exists d, e, \ldots P(a, b, c, d, e, \ldots)$$

(exactly like in Feferman's notation) as

$$C \triangleq \mathcal{I}_2(B, A).$$

And induction over such classes is provided by FS_0 with the axiom

$$\begin{aligned} B \subset X \rightarrow \forall x, y, z[y \in X \rightarrow z \in X \rightarrow \\ (x, y, z) \in A \rightarrow x \in X] \rightarrow \mathcal{I}_2(B, A) \subset X. \end{aligned}$$

§ 3 An informal description of Gentzen's calculus

The version of natural deduction invented by Gentzen, the sequent calculus, combines the virtues of being both practical to use for building derivations, and having good metatheoretic properties. This makes it perfect for the current purpose, since I want a theory in which I can prove theorems, and with which I can also hope to do useful metatheory. In this section I give an informal description of (one form of) the calculus, like one might find in a book.

§ 3.1 The language The language of wffs (well formed formulae) used here is just the $\vee, \neg$ fragment of the language of propositional logic; i.e.

- The atomic propositions P_n are in the language,
- If A is in the language, then $\neg A$ is in the language,
- If A and B are in the language, then $A \vee B$ is in the language.

§ **3.2 The calculus for classical propositional logic** In the following, A and B vary over wffs, Γ and Δ vary over lists of wffs, and a decorated Γ' or Δ'' indicates that it a permuted sublist of the undecorated form. A decorated wff, $A^\dagger$, is called principal.

A sequent is written

$$\Gamma \vdash \Delta,$$

which should be read as 'if all the wffs in Γ are true, then some wff in Δ is true.'

There is one class of axioms, which are called *basic*,

$$\frac{}{A^\dagger, \Gamma \vdash A^\dagger, \Delta}\ basic.$$

Then there are two rules (left, and right — depending on which side of the sequent they affect) for each connective. For negation these are

$$\frac{A^\dagger, \Gamma \vdash \Delta}{\Gamma \vdash \neg A^\dagger, \Delta}\ R\_neg \qquad \text{and} \qquad \frac{\Gamma \vdash A^\dagger, \Delta}{\neg A^\dagger, \Gamma \vdash \Delta}\ L\_neg,$$

and for disjunction,

$$\frac{\Gamma \vdash A^\dagger, B^\dagger, \Delta}{\Gamma \vdash A \vee B^\dagger, \Delta}\ R\_or \qquad \text{and} \qquad \frac{A^\dagger, \Gamma' \vdash \Delta' \quad B^\dagger, \Gamma'' \vdash \Delta''}{\Gamma, A \vee B^\dagger \vdash \Delta}\ L\_or.$$

To this set of rules a structural rule,

$$\frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Delta}\ struct,$$

and a cut rule,

$$\frac{\Gamma' \vdash A^\dagger, \Delta' \quad A^\dagger, \Gamma'' \vdash \Delta''}{\Gamma \vdash \Delta}\ cut.$$

are added.

It should be easy to see that these rules are enough so that for any valid formula A of the fragment of classical propositional logic I use, the sequent

$$\vdash A$$

is provable. The Hauptsatz says that if a formula can be proved using these rules, then it is can be proved without *cut*. The proof Gentzen gives is essentially an algorithm for restructuring any derivation to remove all uses of *cut*.

§ 4 Formalising the calculus

In section § 3 above, I have described a sequent calculus presentation of a fragment of classical propositional logic. That description is enough for an analysis in a book, but I have ignored details and taken things for granted. For instance I assume that you know what a list and what a permutation, or a sublist, of a list is; what rules are and what the closure of a set under a collection of rules is. But if I am to formalise the theory on a machine, I cannot assume any of this; I have exactly the resources that the framework theory, in this case FS_0,

makes available, and all the 'missing' details in the last presentation have to be formalised along with everything else. So the first thing to do is describe the things that I took for granted: wffs, lists of wffs, permutations, etc. (some of this is directly — or almost directly — provided by FS_0; e.g. lists, or closure under a set of rules). In section § 4.1 I will describe these in FS_0, and then in section § 4.2 I will describe the theory itself.

§ 4.1 Preliminaries

the class of wffs The first class to define is that of wffs. The definition I gave in section § 3.1 can be easily translated, since it is already an inductively defined class where it is only ever necessary to appeal to at most two previous members and thus the basic facilities available in FS_0 can be used directly.

The translation from the syntax of section § 3.1 into FS_0 is as follows (I use a pair of square quotation marks $\ulcorner\cdot\urcorner$ to indicate the translation in a readable manner):

$$\begin{aligned} \ulcorner P_n \urcorner &\equiv (\mathit{Prop}, \ulcorner n \urcorner) \\ \ulcorner \neg A \urcorner &\equiv (\mathit{Neg}, \ulcorner A \urcorner) \\ \ulcorner A \vee B \urcorner &\equiv (\mathit{Disj}, (\ulcorner A \urcorner, \ulcorner B \urcorner)), \end{aligned}$$

where '*Prop*', '*Neg*' and '*Disj*' are the names of distinct s-expressions.

After assigning s-expressions to the three names the class of atomic propositions can be defined as

$$\mathit{Atomic} \triangleq \{ (\mathit{Prop}, a) \mid \top \}$$

(i.e. the class of all tuples where the left-hand part is the constant *Prop*, and the right hand part is unconstrained — $\top$ is some tautology). This is a concise version of the equivalent definition

$$\mathit{Atomic} \triangleq \{ (b, a) \mid b = \mathit{Prop} \}.$$

And the rules for negated and disjoint formulae can be defined

$$\begin{aligned} \mathit{Neg\text{–}gen} &\triangleq \{ ((\mathit{Neg}, a), a, a) \mid \top \} \\ \mathit{Or\text{–}gen} &\triangleq \{ ((\mathit{Disj}, (a, b)), a, b) \mid \top \}, \end{aligned}$$

so that the definition of the class of wffs is simply

$$\mathit{Wffs} \triangleq \mathcal{I}_2(\mathit{Atomic}, \mathit{Or\text{–}gen} \cup \mathit{Neg\text{–}gen}).$$

The class of lists Since s-expressions are already available, lists can be defined easily, taking O as the empty list. One rule,

$$\mathit{Wff\text{–}list\text{–}gen} \triangleq \{ ((a, g), g, g) \mid a \in \mathit{Wffs} \},$$

is needed, so that a definition of lists of wffs is

$$\mathit{Wff\text{–}list} \triangleq \mathcal{I}_2(\{O\}, \mathit{Wff\text{–}list\text{–}gen}).$$

Predicates for 'membership' and 'subset' (or 'permuted sublist') are also needed; these have the behaviour that the equivalent *Lisp* functions on lists would have, and can be defined simply as solutions to the equations

$$\begin{aligned} \mathit{Member}(a, O) &= \mathit{False} \\ \mathit{Member}(a, (a, g)) &= \mathit{True} \\ \mathit{Member}(a, (b, g)) &= \mathit{Member}(a, g) \end{aligned}$$

and

$$\begin{aligned} \mathit{Subset}(O, d) &= \mathit{True} \\ \mathit{Subset}(g, d) &= \mathit{And}(\mathit{Member}(\pi_1 g, d), \\ &\qquad \mathit{Subset}(\pi_2 g, d)). \end{aligned}$$

In order to make the presentation more readable, from now on I will abbreviate *Subset* as

$$g \sqsubset d \quad \equiv \quad \mathit{Subset}(g, d) = \mathit{True}.$$

Sequents With facilities for treating primitive lists in hand, sequents can be defined as pairs of lists of wffs; i.e.

$$\ulcorner \Gamma \vdash \Delta \urcorner \quad \equiv \quad (\ulcorner \Gamma \urcorner, \ulcorner \Delta \urcorner)$$

or, formally,

$$\mathit{Seq} \quad \triangleq \quad \{\ (g, d) \mid g \in \mathit{Wff\text{–}list},\ d \in \mathit{Wff\text{–}list}\ \}.$$

§ **4.2 Defining the theory** Now a sequent calculus for classical propositional logic can be formalised as a subclass of *Seq*.

The axioms The class of basic sequents is then defined as

$$\mathit{Basic} \quad \triangleq \quad \{\ ((a, g), (a, d)) \mid ((a, g), (a, d)) \in \mathit{Seq}\ \}.$$

The rules The rules can be defined in pretty much the same way. Since they are all fairly similar, I will give definitions of just three of them (those that I discuss later). The definitions here should be compared with the earlier informal ones.

First, take the most complex of the logical rules: *L_or*. This can be defined as

$$\begin{aligned} \mathit{Lor\text{–}r} \quad \triangleq \quad \{\ & ((((\mathit{Disj}, (a_L, a_R)), g), d), ((a_L, g'), d'), ((a_R, g''), d'')) \mid \\ & ((((\mathit{Disj}, (a_L, a_R)), g), d) \in \mathit{Seq}, \\ & d' \sqsubset d,\ d'' \sqsubset d,\ g' \sqsubset g,\ g'' \sqsubset g\ \} \end{aligned}$$

(notice that the rule needs to check that the goal is a sequent, i.e. that nothing that is not a wff is accidentally included in g or d). Similarly, its dual, *R_or*, can be defined as

$$\begin{aligned} \mathit{Ror\text{–}r} \quad \triangleq \quad \{\ & ((g, ((\mathit{Disj}, (a_L, a_R)), d)), \\ & (g, (a_L, (a_R, d))), \\ & (g, (a_L, (a_R, d)))) \mid \top\ \}. \end{aligned}$$

Notice that this definition takes two identical subderivations; this is because direct use is made of the facilities which FS_0 provides for defining recursively enumerable classes (i.e. the $\mathcal{I}_2(\cdot,\cdot)$ constructor) when the theory itself is formalised. This does not affect the theory though, as will be seen later, it does in a small way affect the way metatheory is done (this quirk has already appeared in the definitions of *Neg–gen* and *Wff–list–gen*).

A final example is the definition of the rule that is to be shown unnecessary: *cut*. This can be defined as

$$\begin{aligned} \mathit{Cut\text{–}r} \quad \triangleq \quad \{\ & ((g,d),(g',(a,d')),((a,g''),d'')) \mid \\ & (g,d) \in \mathit{Seq}, \\ & g' \sqsubset g, d'' \sqsubset d, d' \sqsubset d,\ g'' \sqsubset g\ \}. \end{aligned}$$

The definitions of the rules $(R\_neg, L\_neg, struct)$ are just variations on these patterns.

The theory The theory of sequent calculus for classical propositional logic can now be defined simply as

$$\begin{aligned} \mathit{Logic\text{–}r} &\triangleq \mathit{Rneg\text{–}r} \cup \mathit{Lneg\text{–}r} \cup \mathit{Ror\text{–}r} \cup \mathit{Lor\text{–}r} \\ \mathit{SC} &\triangleq \mathcal{I}_2(\mathit{Basic},\ \ \mathit{Logic\text{–}r} \cup \mathit{Struct\text{–}r} \cup \mathit{Cut\text{–}r}), \end{aligned}$$

With this definition, a sequent $\Gamma \vdash \Delta$ is provable if and only if $\ulcorner \Gamma \vdash \Delta \urcorner \in SC$. It is easy to see that there is an isomorphism between derivations in the system described in section § 3 and derivations in SC.

§ 5 Using *SC*

The class SC is a complete formal specification of the sequent calculus in FS_0, but being formal is not enough: a framework should allow *useable* formalisations of theories, so that proofs of theorems can actually be built, otherwise one could equally use PRA. I have not yet shown, and it is certainly not obvious, that the formalisation given here is usable.

One immediate criticism, for instance, is that it looks as if a lot of work has to be done by every time a rule is applied: e.g. four applications of the *Subset* predicate have to be evaluated at every application of *Lor–r*. Further, this is a function defined inside FS_0 rather than a part of the implementation, so it will be evaluated using a probably not very efficient interpreter, possibly written in a language that is, itself, interpreted or semi-interpreted. This means that the user is working two big steps away from the machine, and so will probably find that building proofs is tiresomely slow.

This sort of problem can be fixed by shifting as much of the work as possible from inside FS_0 to the supporting theorem prover, so that one (and probably, in terms of running time, by far the more costly) level of implementation can be 'stepped around'. This can be done by developing a clutch of lemmas corresponding to rule applications. For instance, a lemma that could be used for

the *Lor–r* rule might look like

$$\begin{array}{l}\forall a, b \in \mathit{Wffs} \forall g, d \in \mathit{Wff\text{–}list} \\ \quad [((a, g), d) \in SC \rightarrow \\ \qquad ((b, g), d) \in SC \rightarrow \\ \qquad\quad (((\mathit{Disj}, (a, b)), g), d) \in SC],\end{array}$$

which, in terms of the abstract version of the theory described in section § 3, is, essentially, the derived rule

$$\frac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta}.$$

Now it is no longer necessary to run any *Subset* tests; the work (once the lemma has been proven) is shifted from the implementation of SC to the lemma and substitution mechanisms of the implementation of FS_0, which might be in, for instance, *RPG*, *Fortran* or *COBOL*, and is (hopefully) much more efficient. Even if it is not efficient, it can probably be made efficient by being carefully rewritten, and such optimisation makes much more sense than optimising any particular piece of FS_0 itself, since it will work equally with any declared theory.

A further point about such collections of lemmas is that they can be extended in any direction; if some special style of proof search is needed, then a set of special lemmas tailored for it can be supplied, and these will be exactly as efficient as basic rules — in a sense the distinction between basic and derived rules has been blurred. In section § 6 I argue that it is practically possible in FS_0 to supply new rules of pretty much arbitrary complexity, derived or even just admissible, this way.

§ 6 Doing metatheory

I now have a formalisation, SC, of classical propositional logic in FS_0 which I can use to build derivations of theorems. This is the first suggested use of FS_0. I still have to show that it can be used for the second: proving theorems *about* a formalised theory. To do this in this section I look at how one might go about formalising one of the basic metatheoretic results of logic in FS_0: Gentzen's Hauptsatz.

§ 6.1 Expressing the theorem The first thing I have to do is to express the theorem, and to do this a version of the theory that does not have the cut rule is needed. This can be defined simply in exactly the same way as SC, but with *cut* deleted, as

$$\mathit{SCCF} \quad \triangleq \quad \mathcal{I}_2(\mathit{Basic}, \mathit{Logic\text{–}r} \cup \mathit{Struct\text{–}r})$$

and it is easy to see that

$$\mathit{SCCF} \quad \subset \quad SC.$$

The other direction,

$$SC \quad \subset \quad \mathit{SCCF}, \tag{1}$$

is the Hauptsatz, and the rest of this section discusses its proof.

§ **6.2 An overview of the proof** Proofs of the Hauptsatz are easy to find in books on proof theory, for instance, Girard gives a detailed discussion in [10, §2.3], though the presentation in [20] is closer to that here. Those proofs however, cannot be directly translated into FS_0, since they make use of transfinite induction principles which it does not have. Alongside these proofs are, often, remarks to the effect that a proof in primitive recursive arithmetic *is* possible [10, §2.3.11], but not practical; Girard, for instance, describes the details as '... straightforward, but terribly long'. And since the proof is not practical, it is not given, only sketched. Part of the intended purpose of FS_0, which has only the same induction as primitive recursive arithmetic, is to make such proofs practical (even to the extent that they can be put on a machine).

Since I cannot use transfinite induction to prove the theorem directly, I have to find another, more indirect, way. I do this by mapping, inside FS_0, the usual transfinite ordering onto the natural numbers using a primitive recursive function, so that that part of the ordering that is needed for the proof is preserved (i.e. the mapping is not monotonic — that is not possible — but it suffices for the current purpose). I can then use the values that result from this as a 'complexity measure' to label derivations. I also ensure that the complexity of a derivation is 0 if and only if it does not use *cut*. Then I can show, using simple induction on the structure of derivations, that a derivation can be transformed into another with a lower complexity (unless the complexity is already 0) and, given this, that a derivation can be progressively transformed into another with a complexity of 0. The other parts of the proof, i.e. the transformations performed on a derivation, are the same as one would need for a proof using ordinary transfinite induction.

Overview of the rest of this section The rest of this section is structured as follows: in section § 6.3 I define the natural numbers, in section § 6.4 I define a measure of complexity on derivations, in section § 6.5 I show how to prove the central lemma of complexity reduction for proofs, and finally in section § 6.6 I quickly put everything together to get the main result.

§ **6.3 The natural numbers** Since I will measure the complexity of derivations with natural numbers, the first thing I have to do is define them. A number can be represented by a list that long; i.e.

$$\begin{aligned} \ulcorner 0 \urcorner &\equiv O \\ \ulcorner s(n) \urcorner &\equiv (O, \ulcorner n \urcorner), \end{aligned}$$

so that a formal definition is just

$$\begin{aligned} \mathit{Nat\text{–}gen} &\triangleq \{\ ((O, n), n, n)\ \} \\ \mathit{Nat} &\triangleq \mathcal{I}_2(\{O\}, \mathit{Nat\text{–}gen}). \end{aligned}$$

I also need an ordering relation and a function for addition; these can be defined as solutions to the sets of equations

$$\begin{aligned} \mathit{Plus}(m, O) &= m \\ \mathit{Plus}(m, n) &= (O, \mathit{Plus}(m, \pi_2 n)) \end{aligned}$$

and

$$\begin{aligned} \mathit{Less}(a, O) &= \mathit{False} \\ \mathit{Less}(n, (O, n)) &= \mathit{True} \\ \mathit{Less}(n, m) &= \mathit{Less}(m, \pi_2 n). \end{aligned}$$

Like *Subset* earlier, for the sake of readability I will abbreviate *Less* as

$$m < n \quad \equiv \quad \mathit{Less}(m, n) = \mathit{True}.$$

§ **6.4 Attaching a measure to a derivation** A simple measure of the complexity of a derivation in an (inductively defined) class is just the number of steps in it. Given any class

$$C \quad \triangleq \quad \mathcal{I}_2(B, A),$$

a new class C_C can be defined as

$$\begin{aligned} B_C &\triangleq \{\ (x, \ulcorner 1 \urcorner) \mid x \in B\ \} \\ A_C &\triangleq \{\ ((x, l), (x', l'), (x'', l'')) \mid \\ &\qquad (x, x', x'') \in A, \\ &\qquad l = \mathit{Plus}(\mathit{Plus}(l', l''), \ulcorner 1 \urcorner)\ \}. \\ C_C &\triangleq \mathcal{I}_2(B_C, A_C) \end{aligned}$$

and it is easy to see that

$$x \in C \quad \leftrightarrow \quad \exists l[(x, l) \in C_C],$$

where l is the number of steps taken in some derivation of x.

Such a simple measure cannot be used here; something more complicated is needed. As I said earlier, the usual way to prove the theorem is with transfinite induction, and the well-ordering needed is a lexicographical ordering on the pair $\langle D, S \rangle$, where S is a measure of the size of the derivation and D is a measure of the 'cut degree' (the complexity of the cut formula). I can map this pair into into the natural numbers as

$$\langle D, S \rangle \quad \mapsto \quad 2^{2^{\cdot^{\cdot^{\cdot^{2^S}}}}} \Big\}_{D+1}$$

(i.e., a column D high of 2s). And this can easily be defined in FS_0 (using primitive recursion on D) as the function

$$N\text{–}Ex(\ulcorner D \urcorner, \ulcorner S \urcorner).$$

Now a working measure of complexity can be associated with derivations, like in the example above. In fact a pair of numbers $m = (c, l)$ is used, but the complexity is measured just by c; the other number, l, is the size of the derivation.

First, I define the way the measure works with the 'logical' rules. This can be done by defining classes as

$$\begin{array}{rcl} \mathit{M\text{–}Base} & \triangleq & \{\ (\ulcorner 0 \urcorner, \ulcorner 1 \urcorner)\ \} \\ \mathit{M\text{–}Step} & \triangleq & \{\ ((c,l),(c',l'),(c'',l'')) \mid \\ & & \qquad l = \mathit{Plus}(\mathit{Plus}(l',l''),\ulcorner 1 \urcorner), \\ & & \qquad c = \mathit{Cr1}((c',l'),(c'',l''))\ \}, \end{array}$$

where

$$\begin{array}{rcl} \mathit{Cr1}((\ulcorner 0 \urcorner,l'),(\ulcorner 0 \urcorner,l'')) & = & \ulcorner 0 \urcorner \\ \mathit{Cr1}((c',l'),(\ulcorner 0 \urcorner,l'')) & = & (O,\mathit{Plus}(c',l'')) \\ \mathit{Cr1}((\ulcorner 0 \urcorner,l'),(c'',l'')) & = & (O,\mathit{Plus}(l',c'')) \\ \mathit{Cr1}((c',l'),(c'',l'')) & = & (O,\mathit{Plus}(c',c'')). \end{array}$$

Then the 'logical' rules are extended as

$$\begin{array}{rcl} \mathit{Basic}_C & \triangleq & \{\ (s,m) \mid s \in \mathit{Basic}, m \in \mathit{M\text{–}Base}\ \} \\ \mathit{Logic\text{–}r}_C & \triangleq & \{\ ((s,m),(s',m'),(s'',m'')) \mid \\ & & \qquad (s,s',s'') \in \mathit{Logic\text{–}r} \cup \mathit{Struct\text{–}r}, \\ & & \qquad (m,m',m'') \in \mathit{M\text{–}Step}\ \}. \end{array}$$

Here *Struct–r* has been added to the logical rules for the sake of convenience. With the definitions so far c, the complexity measure, will not take any value other than $\ulcorner 0 \urcorner$. If it ever takes a value other than $\ulcorner 0 \urcorner$ then it will grow, but so long as only the 'logical' rules and *basic* are used, this cannot happen. This is exactly the right behaviour, since a derivation that does not use *cut* will thus have an complexity measure of $\ulcorner 0 \urcorner$.

I want to use c to track information about cuts in the derivation, and the modified form of the cut rule has not been defined yet. Before doing so, a function that measures the number of connectives in a formula is needed, and this is defined as a solution to the equations

$$\begin{array}{rcl} \mathit{Comp}(\ulcorner P_n \urcorner) & = & \ulcorner 0 \urcorner \\ \mathit{Comp}(\ulcorner \neg A \urcorner) & = & (O,\mathit{Comp}(\ulcorner A \urcorner)) \\ \mathit{Comp}(\ulcorner A \vee B \urcorner) & = & (O,\mathit{Plus}(\mathit{Comp}(\ulcorner A \urcorner),\mathit{Comp}(\ulcorner B \urcorner))). \end{array}$$

Then the new form of the cut rule can be defined as

$$\begin{array}{rcl} \mathit{Cut\text{–}r}_C & \triangleq & \{\ (((g,d),(c,l)),((g',(a,d')),(c',l')),(((a,g''),d''),(c'',l''))) \mid \\ & & \qquad (g,d) \in \mathit{Seq}, \\ & & \qquad g' \sqsubset g,\ d'' \sqsubset d,\ d' \sqsubset d,\ g'' \sqsubset g, \\ & & \qquad c = \mathit{Cr2}((c',l'),(c'',l''),\mathit{Comp}(a)), \\ & & \qquad l = \mathit{Plus}(\mathit{Plus}(l',l''),\ulcorner 1 \urcorner)\ \}, \end{array}$$

where

$$\begin{array}{rcl} \mathit{Cr2}((\ulcorner 0 \urcorner,l'),(\ulcorner 0 \urcorner,l''),n) & = & \mathit{N\text{–}Ex}(n,\mathit{Plus}(l',l'')) \\ \mathit{Cr2}((c',l'),(\ulcorner 0 \urcorner,l''),n) & = & \mathit{N\text{–}Ex}(n,\mathit{Plus}(c',l'')) \\ \mathit{Cr2}((\ulcorner 0 \urcorner,l'),(c'',l''),n) & = & \mathit{N\text{–}Ex}(n,\mathit{Plus}(l',c'')) \\ \mathit{Cr2}((c',l'),(c'',l''),n) & = & \mathit{N\text{–}Ex}(n,\mathit{Plus}(c',c'')) \end{array}$$

So the definition of SC_C is

$$SC_C \triangleq \mathcal{I}_2(Basic_C, Logic\text{–}r_C \cup Cut\text{–}r_C),$$

and I can show that

$$x \in SC \leftrightarrow \exists m[(s,m) \in SC_C]. \tag{2}$$

§ **6.5 Reducing the Complexity of a derivation** Now I prove the lemma,

$$\begin{aligned}(s,(c,l)) \in SC_C \rightarrow (s,(\ulcorner 0 \urcorner, l)) \in SC_C \vee \\ \exists c_1, l_1[c_1 < c \wedge l_1 < c \wedge (s,(c_1,l_1)) \in SC_C],\end{aligned} \tag{3}$$

which is essentially the theorem itself; however, in order to do this I need to use induction, and induction is not available over such formulae in FS_0. The solution is to define a a class,

$$\begin{aligned}SC^*_C \triangleq \{\ (s,(c,l)) \mid (s,(\ulcorner 0 \urcorner, l)) \in SC_C \vee \\ \exists c_1, l_1[c_1 < c \wedge l_1 < c \wedge (s,(c_1,l_1)) \in SC_C]\ \},\end{aligned}$$

which is equivalent to the consequent of (3) and then then it is possible to prove the equivalent statement:

$$SC_C \subset SC^*_C.$$

The base case The base case is trivial. I have to show that

$$s_C \in Basic_C \rightarrow s_C \in SC^*_C,$$

and can argue thus:

$$\begin{aligned}s_C \in Basic_C &\rightarrow s_C = (s,m) \wedge s \in Basic \wedge m \in M\text{–}Base \\ &\qquad\qquad [\text{for some } s \text{ and } m] \\ &\rightarrow m = (\ulcorner 0 \urcorner, \ulcorner 1 \urcorner) \\ &\rightarrow (s,(\ulcorner 0 \urcorner, \ulcorner 1 \urcorner)) \in SC_C \\ &\rightarrow s_C \in SC^*_C.\end{aligned}$$

The step case This is more complicated. I have to show that, given

$$s'_C \in SC^*_C, \tag{4}$$

$$s''_C \in SC^*_C, \tag{5}$$

$$(s_C, s'_C, s''_C) \in Logic\text{–}r_C \cup Cut\text{–}r_C, \tag{6}$$

it follows that

$$s_C \in SC^*_C.$$

I can do this by analysing a hierarchy of cases. The transformations are well known, and there seems little point in describing all of them, so I will give the general procedure, and then discuss one of the most complicated cases in detail; the others follow the same pattern.

By (4) and (5), given $s'_C = (s,(c,l))$ and $s''_C = (s'',c'',l''))$, there are $(s',(c'_1,l'_1))$ and $(s'',(c''_1,l''_1))$ in SC_C and either one of c' and c'' is an encoded natural number other than $\ulcorner 0 \urcorner$ or both of them are $\ulcorner 0 \urcorner$.

Assuming the former, then either $c'_1 < c'$ and $l'_1 < c'$, and $c''_1 \leq c''$ and $l''_1 \leq c''$ (depending on whether or not $c'' = \ulcorner 0 \urcorner$), or vice versa. By (6) I can again separate into two subcases. If

$$(s_C, s'_C, s''_C) \in \mathit{Logic}\text{–}r_C,$$

then it is easy to show that

$$((s,(c_1,l_1)),(s',(c'_1,l'_1)),(s'',(c''_1,l''_1)) \in \mathit{Logic}\text{–}r_C,$$

where

$$\begin{aligned} c_1 &= \mathit{Cr1}((c'_1,l'_1),(c''_1,l''_1)) \\ l_1 &= (O, \mathit{Plus}(l'_1,l''_1)) \end{aligned}$$

and to show that $s_C \in SC^*_C$ I have only to show that $c_1 < c$ and $l_1 < c$, which is a matter of simple manipulation.

Alternatively, if $(s_C, s'_C, s''_C) \in \mathit{Cut}\text{–}r_C$, then the argument follows the same pattern, except that the new values are

$$\begin{aligned} c_1 &= \mathit{Cr2}((c'_1,l'_1),(c''_1,l'')) \\ l_1 &= (O, \mathit{Plus}(l'_1,l''_1)). \end{aligned}$$

Now consider when both c' and c'' are $\ulcorner 0 \urcorner$. Again, by (6) there are two subcases: (s_C, s'_C, s''_C) is in either $\mathit{Logic}\text{–}r_C$ or $\mathit{Cut}\text{–}r_C$. In the first case it is easy to show that c is also $\ulcorner 0 \urcorner$. The cut rule is the interesting case — then the final derivation of $s_c = ((g,d),(c,l))$ has to be of the form

$$\frac{((g',(a,d')),(\ulcorner 0 \urcorner,l'))\quad(((a,g''),d''),(\ulcorner 0 \urcorner,l''))}{((g,d),(c,l))}\ \mathit{Cut}\text{–}r_C,$$

where

$$\begin{aligned} &g' \sqsubset g,\ d'' \sqsubset d,\ d' \sqsubset d,\ g'' \sqsubset g, \\ &c = \mathit{Cr2}((\ulcorner 0 \urcorner,l'),(\ulcorner 0 \urcorner,l''),\mathit{Comp}(a)), \\ &l = (O,\mathit{Plus}(l',l'')). \end{aligned}$$

The proof proceeds by analysing the ways that $((g',(a,d')),(\ulcorner 0 \urcorner,l'))$, or more briefly s'_C, might have been derived. Most of these cases are very similar, so only one (the most complicated, and the reason why transfinite induction is used) will be considered here. First note that, because $c' = \ulcorner 0 \urcorner$, s'_C cannot have been derived using $\mathit{Cut}\text{–}r_C$, and that there is a case for each of the other original rules of SC, i.e. *basic*, $\mathit{Lor}\text{–}r$, etc.

Consider the case where s'_C is derived by $\mathit{Ror}\text{–}r$, i.e. $a = ((\mathit{Disj},(a_L,a_R))$ and $((g',((\mathit{Disj},(a_L,a_R)),d'));\mathit{then}(\ulcorner 0 \urcorner,l'))$ is shown to be in SC_C by

$$\frac{((g',(a_L,(a_R,d'))),(\ulcorner 0 \urcorner,l''))\quad((g',(a_L,(a_R,d'))),(\ulcorner 0 \urcorner,l''))}{((g',((\mathit{Disj},(a_L,a_R)),d')),(\ulcorner 0 \urcorner,l'))}\ \mathit{Ror}\text{–}r.$$

Then there are two possible ways that $(((a, g''), d''), (\ulcorner 0 \urcorner, l''))$ could have been derived: either by a thinning or 'right' rule, or by *Lor–r* — it is not possible for *Lneg–r* to derive a sequent with a disjunction as principle formula. Consider the case when *Lor–r* was the rule used; then the principle formula is $a' = a$; i.e. the derivation is:

$$\frac{(((a_L, g'''), d'''), (\ulcorner 0 \urcorner, l''')) \quad (((a_R, g'''''), d'''''), (\ulcorner 0 \urcorner, l'''''))}{((((Disj, (a_L, a_R)), g''), d''), (\ulcorner 0 \urcorner, l''))} \; Lor\text{–}r,$$

where

$$g''' \sqsubset g'',\ d''' \sqsubset d'',\ g''''' \sqsubset g'',\ d''''' \sqsubset d''.$$

An alternative derivation for s_C is

$$\frac{((g, (a_R, d)), (c_1', l_1')) \quad (((a_R, g'''''), d'''''), (\ulcorner 0 \urcorner, l'''''))}{((g, d), (c_1, l_1))} \; Cut\text{–}r_C,$$

where

$$\frac{((g', (a_L, (a_R, d'))), (\ulcorner 0 \urcorner, l''')) \quad (((a_L, g'''), d'''), (\ulcorner 0 \urcorner, l'''))}{((g, (a_R, d)), (c_1', l_1'))} \; Cut\text{–}r_C$$

Then all that is left to be done is to check that these are proper derivations and that $c_1 < c$ and $l_1 < c$, which is not hard.

§ **6.6 Eliminating the cuts** Now, with lemma (3), it is an easy matter to prove that

$$(s, (a, l)) \in SC_C \quad \rightarrow \quad \exists l[(s, (\ulcorner 0 \urcorner, l)) \in SC_C] \tag{7}$$

and, finally, that

$$(s, (\ulcorner 0 \urcorner, l)) \in SC_C \quad \rightarrow \quad s \in SCCF. \tag{8}$$

Then by composing (2), (7) and (8), the theorem, (1), follows.

§ 7 Using Metatheory

With a proof of the Hauptsatz available, the next question is: 'of what use is the result to practical people?'[1].

The most immediate application is to show the consistency of SC; given (1) it is not hard to show that not every sequent is derivable in SC; e.g.

$$\ulcorner \vdash \neg(A \vee \neg A) \urcorner \notin SC,$$

[1]Strictly speaking, the answer to this question is: 'not much, since no one, in practice, is interested in propositional logic, and anyway the various corollaries discussed here all have, for the propositional case, easier proofs that do not need the Hauptsatz.' However scaling the proof up for predicate logic, where the same results much more usefully hold, is not hard; the problem of dealing with bound variables in predicate logic is an independent, though large, problem — bound variables do not introduce anything new into the proof presented here, though they do make case analysis quite a bit bigger. The issue of how to treat bound variables in FS_0 is explored in [13] and [14].

since if a formula is in SC it is in $SCCF$, and an analysis of cases shows that this formula cannot be derived in $SCCF$.

This proof of the consistency is one example of a range of practical corollaries of the result. The most important after consistency is probably either Herbrand's theorem or the interpolation theorem [6]. The commonest justification for proof development systems is formal software verification, an activity which needs particularly powerful tools for structuring and combining collections of theories. The interpolation theorem is precisely the tool needed to track the relationships in such collections.

Showing the consistency of the implementation is something that can be done without considering the structure of the proof of the Hauptsatz itself, however an important point about the proof is that it is constructive. This is significant if, for instance, it is used to build a proof of the interpolation theorem, since one might then actually want to use the proof to extract interpolants. Unfortunately there are problems with this: if the proof here is considered as a program it is not a very good one — the problem is not the size of the complexity measure, which is a fairly accurate assessment[2] of the upper bound on the computation (and the bound is enormous — far outside the bounds of feasible computation — though it should be remembered that it is for a worst case), but that it is 'sloppily coded', i.e. from a programmer's point of view it does not do as much work on each iteration as it might, and it has bad normalisation properties. However these are programming rather than mathematical issues and (the second at least) can be addressed by the programmer's slogan, 'get it right, then make it fast' — a proof in hand can always be optimised if and as necessary.

§ 8 Related work

Related work divides into several parts: there is complementary work that has been done in FS_0, and there is work on frameworks, and on doing, and using, metatheory in other proof development systems.

§ 8.1 Using FS_0 as a framework This paper can be thought of as complementary to [14], where a full sequent calculus of first order logic is presented and the problems of working on a machine are discussed in more detail. In that paper a much simpler metatheorem (the existence of equivalent prenex normal forms for formulae) is discussed — the intention of the paper is rather to look at how it is possible to work with a formalised binding mechanism. The issues are also examined in my thesis [13]. So far as I, or Feferman, knows, this is the only work which has been done on practical applications of FS_0 as a framework.

§ 8.2 Other work similar to FS_0 The approach of FS_0 can be traced back to the work of Post in the thirties. He was the first to formalise the idea of a proof system as a recursively enumerable set, and his work was built on by Smullyan in the sixties. Their intention, however, was to capture the idea of a derived formula; they did not really look at the possibility of doing proof theory in their proposed systems, even in theory. Gödel, who had slightly different concerns at the time, must, for his incompleteness results, have been

[2] At least for predicate logic — see the previous footnote.

the first to do this. The first suggestion that such a theory might be actually be used, as the basis of a mechanical proof development system, was by Davis and Schwartz [7], but their paper does not convincingly address the practical issues involved.

More recently Basin and Constable, in [2], have suggested an approach that is in many ways similar to what is described here: the chief difference is that they define a logic in terms of an abstract data type, rather than building it explicitly, and they do not concern themselves particularly with the logic used to treat this data type — they happen to use a type-theoretic approach (with abstract data types implemented as Σ-types) but this seems to be incidental, they insist only on a constructive metatheory sufficiently powerful to support such abstraction. They also point out that the FS_0 approach can be directly formulated in their system using inductive types. This is true, but ignores one intended purpose of FS_0, which is to give a simple, finite, description of what exactly a formal system is. Also, as is pointed out below in section § 9, while the enormous proof theoretic strength of their supporting system is useful, it does not seem to be necessary for most purposes. However, one definite advantage of using abstract data types is that there are no 'quirks' like those mentioned, for example, in the definition of *Ror–r* above.

§ **8.3 Research in type-theoretic frameworks** The largest body of work on theories suitable for use in framework proof development systems must be that based on type theories. This work goes back to the *Automath* project, which is surveyed in [8], and details of the theories that were used can be found in [24]. More recently, work at Edinburgh has built on this with the 'Edinburgh Logical Framework' (also called *LF*); details of the theory of this can be found in [11], and a collection of worked examples is described in [1].

The idea is to exploit the idea of an isomorphism between propositions and types, so that terms inhabiting a type are isomorphic to proofs of the corresponding proposition. The great advantage of this is that substitution for terms and formulae comes practically for free, since there is already a general substitution mechanism available in the lambda calculus facilities that come with the type theory. This is a very flexible approach, but it does have some problems: it is not always possible to take a presentation as given and encode it directly and intuitively in the *LF*; skill, and knowledge of proof theory may be needed, and the resulting encoding may not obviously correspond to the original presentation.

There is also the fact that the *LF* is not very good for doing metatheory: it cannot deal with the notion of an admissible rule (though simple derived rules are certainly possible). However it is not really fair to criticise the system for this, since it was developed with different concerns in mind. In particular the type theory it is based on was deliberately chosen to be as weak as possible so that it would be easier (or even just become possible) to develop various sorts of uniform proof procedures — it is intended to be used with tactics rather than metatheory; see, for instance, [19]. This does not mean that it is not possible to do general metatheory in any *LF* style system. *LF* is a fragment of the very powerful Calculus of Constructions, it is possible to move an encoding

(carefully) into this and make use of the more powerful facilities there to develop metatheoretic results, such as admissible rules —- in fact Taylor works through some small examples of this approach in [23], where he develops a pair of verified tactics for the theory of semigroups. Pollack, in unpublished work, has also considered a proof of the deduction theorem (this is also done with FS_0 in [13]). Similar ideas are implicit in the work of Basin and Howe [3] which looks at how to use a (very powerful) Martin-Löf style type theory as a *LF* style framework.

Other work that uses lambda calculus style frameworks, but does not use the notion of propositions as types, can be found also in work on *Isabelle* [18], and *Lambda Prolog* [16], systems based on higher order intuitionistic logic.

Clearly then, it is a matter of choice and circumstance whether a type-theoretic, or Post-style, framework is suitable for some particular piece of work.

§ **8.4 Miscellaneous work** There is also other less classifiable work that should be mentioned. In [5], Boyer and Moore develop, and describe the implementation of, a metatheorem as an extension to their theorem prover, *Nqthm*. The work is very similar to the example explored in section § 1; their system relies on a uniform proof procedure supplemented with a powerful lemma facility, and they look at how it can be modified so that a whole class of lemmas characterised by a single metatheorem can be added, so as to avoid having to add, piecemeal, each instance that is needed to prove some particular theorem. They do not do this by using a framework logic, but by taking advantage of the fact that their theorem prover is implemented in a system that is an extension of the system it proves theorems about. Thus they use the system to prove the theorem and then 'reflect' (they do not use the word) it into the implementation. This is interesting, not only because it is a practical example of using metatheory to extend a theorem prover, but also because it works with a logic that is, in many ways, similar to FS_0 in that it resembles *Lisp* restricted to primitive recursive functions[3].

Another example of metatheory in *Nqthm* is the proof, by Shankar [21], of the Church-Rosser theorem. The distinction here is that this is metatheory as mathematics for its own sake, rather than as a means for making work in the object theory easier. However it is one of the most substantial metatheoretic results that has been formalised and machine checked. Also, similar work done by Berardi in a *LF*-style framework is described in [4].

§ 9 Conclusions

The point of this paper is to try to show that FS_0 does what Feferman claims it should; it provides a simple and flexible characterisation of the idea of a formal theory, and it can be used as a framework in which it is practically possible to formalise a theory, prove theorems of that theory, and also prove substantial metatheorems about it — especially metatheorems that might help a user in proving further theorems in the theory.

§ **9.1 Doing metatheory** As I said earlier, Girard mentions in passing that a proof of the Hauptsatz in *PRA* is possible but impractically long. And this

[3]Since then, however, *Nqthm* has been extended with a more powerful induction principle using ordinal notations, though so far as I know this has not as yet been needed.

is after he has developed, in considerable detail, the machinery of a Gödel encoding of the sequent calculus, etc. The proof sketched here is developed from first principles in a theory which is a conservative extension of PRA and is a practical proposition for machine proof. (Admittedly, I only treat propositional logic here, not full predicate logic, but the proof can easily be extended, and as argued in [14], a binding mechanism can be dealt with in FS_0). One can also argue that the Hauptsatz is almost a 'one off' piece of metatheory: the amount of effort needed to prove it is exceptional, since a lot of the other results that one might want are corollaries of it. In particular, many metatheorems corresponding to admissible rules become available with little more work.

One might wonder if the proof would have been easier if I had not been restricted to using only Σ^0_1-induction, and clearly this does make things slightly more complex than they would otherwise be, e.g. there would be no need to worry about giving an explicit bound on the induction. But an explicit bound is a useful thing to have, since it provides a measure of the complexity of an algorithm (and in this case shows that in general it is far outside the limits of what is practically computable). If it was felt to be necessary, however, there are simple ways to extend FS_0 to allow much stronger induction principles — though at the cost of non-finitist proofs.

There are some problems with using FS_0 that perhaps cannot be properly fixed: any attempt to interpret the resulting proofs as programs is going to find that they do not have very good normalisation properties, simply because of the nature of a primitive recursive/inductive theory. There are some ways that the theory could be extended to improve the problem, but perhaps only at the cost of damaging it in other ways by making it much more complex, and such aesthetic considerations are practically important for instance, for pedagogic reasons (another intention behind the theory) or if one were to try, as Feferman has suggested, to prove the second incompleteness theorem.

§ **9.2 Further work** There are many possible directions of further work. My current intention is to implement a version of Talcott's theory of binding structures [22]. This is a sufficiently general framework so that it should be easily reusable for any particular theory that a user wants to implement and would provide an effective answer to the criticism that there is no facility for handling substitution in FS_0. Beyond that I have no specific plans. The obvious project is a full machine checked proof of cut elimination, and an exploration of some of its corollaries, such as the interpolation theorem. The other possibility is a machine checked proof of the second incompleteness theorem, the other basic result of syntactic metatheory. This would be interesting for itself, and the machinery that would have to be developed along the way would, I believe, be independently useful.

Acknowledgements

I would like to thank Alan Smaill and Solomon Feferman for helpful criticism and comments. Paul Taylor's code for setting proofs was used.

References

[1] Arnon Avron, Furio Honsell, Ian Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–352, 1992.

[2] David Basin and Robert Constable. Metalogical frameworks. In [12].

[3] David Basin and Doug Howe. Some normalization properties of Martin-Löf's type theory, and applications. In Takayasu Ito and Albert R. Meyer, editors, *TACS '91, Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 475–494. Springer, Berlin, 1991.

[4] Stefano Berardi. Girard normalisation proof in LEGO. In [12].

[5] Robert Boyer and J. Strouther Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In Robert Boyer and J. Strouther Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, New York, 1981.

[6] William Craig. Linear reasoning. A new form of the Herbrand-Genzen theorem. *Journal of Symbolic Logic*, 27:250–268, 1957.

[7] Martin Davis and Jacob T. Schwartz. Metamathematical extensibility for theorem verifiers and proof-checkers. *Computers and Mathematics with Applications*, 5:217–230, 1979.

[8] N. G. de Bruijn. A survey of the project Automath. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, New York, 1980.

[9] Solomon Feferman. Finitary inductive systems. This volume.

[10] Jean-Yves Girard. *Proof Theory and Logical Complexity, volume 1*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Naples, 1987.

[11] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[12] Gérard Huet and Gordon Plotkin, editors. *Logical Environments*. Cambridge University Press, Cambridge, 1993.

[13] Seán Matthews. *Metatheoretic and Reflexive Reasoning in Mechanical Theorem Proving*. PhD thesis, University of Edinburgh, 1992.

[14] Seán Matthews, Alan Smaill, and David Basin. Experience with FS_0 as a framework theory. In [12].

[15] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *Lisp 1.5 Programmer's Manual*. M.I.T. Press, Cambridge, Massachusetts, 1965.

[16] Dale Miller. Abstractions in logic programs. In [17].

[17] Piergiorgio Odifreddi, editor. *Logic and Computer Science*, volume 31 of *APIC Studies in Data Processing*. Academic Press, London, 1990.

[18] Larry Paulson. Isabelle: the next 700 theorem provers. In [17].

[19] David J. Pym and Lincoln Wallen. Proof-search in the $\lambda\Pi$-calculus. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 309–340. Cambridge University Press, Cambridge, 1991.

[20] Helmut Schwichtenberg. Proof theory: Some applications of cut-elimination. In Jon Barwise, editor, *The Handbook of Mathematical Logic*, chapter D2. North-Holland, Amsterdam, 1977.

[21] Natarayan Shankar. A Mechanical Proof of the Church-Rosser Theorem. *Journal of the ACM*, 35:475–522, 1988.

[22] Carolyn Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112:99–143, 1993.

[23] Paul Taylor. Using constructions as a metalanguage. Technical Report ECS–LFCS–88–70, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, December 1988.

[24] D. T. van Daalen. *The language theory of Automath*. PhD thesis, Eindhoven University of Technology, 1980.