Implementing FS_0 in Isabelle: adding structure at the metalevel

Seán Matthews
Max-Planck-Inst., Im Stadtwald
66123 Saarbrücken, Germany
<sean@mpi-sb.mpg.de>

April 30, 1995

ABSTRACT Often the theoretical virtue of simplicity in a theory does not fit with the practical necessities of working with it. We present as a case study an implementation in a generic theorem prover (Isabelle) of a theory (FS_0) which at first sight seems to lake all the facilties needed to be practically usable. However, we show that we can use the facilties available in Isabelle to provide all the structuring facilities (modules, abstraction, etc.) that are needed without compromising the simplicity of the original theory in any way, resulting in a thouroghly practical implementation. We further argue that it would be difficult to build a custom implemenation as effective.

§1 INTRODUCTION

A great many logics have been proposed as tools in computer science, especially for all sorts of formal, machine checked reasoning. However, if we try to implement these theories in some practical manner, we find that what has been proposed by theoreticians as a practical tool has to be augmented in all sorts of ways before it really becomes a practical tool. Essentially, the basic tools of structured programming and other facilties need to be imported. Unfortunately, this means that a proof theory which originally could be summarised on a page or so, grows to fill a manual, and is augmented with descision procedures an other extras, which can be difficult to verify.

A suggested solution to this problem is what have become known as 'Logical frameworks': systems designed to be suitable for implementing a wide range of different logics easily, so that they can be presented in a uniform manner to users, allowing the same theorm proving facilties to be reused across a wide range of implemented theories, instead of having to be rebuilt from scratch each time. The basic idea of a logical framework (we will be concentrating in particular here on the Isabelle system) is to make it easier to implement logics, via some sort of equation of the Syntax + Axioms + Rules = Theorem prover form; but in fact we get more than that. Because a logical framework based system is generic, its implementors can afford to invest in much more powerful facilties, since they are likely to be reused in a range of different contexts (in fact, since they don't know what those contexts will be, they have to provide powerful and general tools so as to improve the possibility that they are usable any particular context). But since this machinery is developed prior to the implementation of any particular theory it must be independent of the details of such theories. Thus given an initial proposal for a theory to be implemented, a logical framework based system should not only be able to provide a much quicker implemenation (via the equation given above) but also might be able to provide some, or all of the structuring facilities that are needed for practical proof development, so that they are actually independent of the theory to implemented. If this is so, then the theory itself does not have to be extended, and thus can be implemented in a way that is closer to the original proposal.

Feferman's FS_0 [3] is such a theory, in this case as nominated as a suitable vehicle for machine checked metatheory. While it is simple (20 or so axioms in a three sorted first-order logic), it so primitive that it is not at all clear that it is practically usable (especially in the form that Feferman gives it). It comes with none of the structuring facilities which we usually depend on when developing large theories (modularity, abstraction, etc.), but these have to be provided somehow in the implementation. We show how we can we can get these directly from Isabelle.

§1.1 *Contributions* We see this paper as making the following contributions. We show the effectiveness of a generic theorem prover such as Iabelle for dealing with an unusual logic and how its facilties can be exploited to provide a great deal of high-level structuring of development in such a theory, without having to introduce such structuring facilties into the logic to be implemented itself, and thus complicating it unnecessarily. We also claim that this case example is an argument that implementation in a generic system can in the end sometimes be more effective than a custom implementation, since so many of the facilties our implementation provides are exploit the powerful facilities that a generic system has to provide (in fact the design is directly driven by the facilities Isabelle provides).

Secondly, and independently We demonstrate that FS_0 is a plausible theory for real computer supported theory development, by presenting the first practical implementation[1]

§1.2 *Outline of paper* The outline of the rest of this paper is as follows: In §2 and §3 we briefly describe FS_0 as background, in §4 we discuss the facilities that we want to provide for theory definition in our implementation, in §5 we describe how we have provided these, in §6 we describe some of the tools for proving theorems in FS_0, in §7 we then briefly outline some of the theory development we have performed, and in §8 we present our conclusions.

§2 The theory FS_0

FS_0 is a theory in the tradition of Gödel's incompleteness results: one can think of it as a 'rational reconstruction' of the results of the preliminary development that that Gödel did in arithmetic (i.e. building tools for doing 'gödel-numbering') to prove his theorems, and is a conservative extension of primitive recursive arithmetic. The details can be found in [3].

A first impression is that the theory is very simple, and, as we have said, there are various reasons for not adulterating that simplicity, so it should be implemented pretty much as it stands: as a three-sorted classical first order, finitely axiomatisable theory of s-expressions, primitive recursive functions and recursively enumerable classes, that resembles Pure Lisp. A summary outline of Feferman's definition is as follows:

- There is the sort S of s-expressions. This is contains a leaf object nil, and is closed under a pairing function $(\_,\_)$; equality is defined in the obvious way over s-expressions.
- There is the sort F of functions. All functions are of the form $S \rightarrow S$ and function application is denoted by ‘, so that if f is a function and t is an s-expressions then $f`t$ is a function application of sort S. we have a small set of basic functions on s-expressions, Id (identity), π_1, π_2 (car and cdr), and Dec (decide). Most of these should be well known, apart, perhaps, from the last which behaves as follows:

  $$Dec`(((a,b),c),d) = \begin{cases} a = b & \rightarrow c \\ a \neq b & \rightarrow d \end{cases}$$

  There are also constant functions $K(a)$ of sort $S \rightarrow S$, where $K(a)`b = a$.
  The basic functions can be combined using combinators of the form $F \times F \rightarrow F$, of which there are three, as follows. Composition, $\_ * \_$, where $(f * g)`t = f`(g`t)$; pairing, $[\_,\_]$, where $[f,g]`t = (f`t, g`t)$; and structural recursion, $Rec(\_,\_)$, where,

[1] So far as we, or Feferman, know.

if $h = Rec(f, g)$,

$$\begin{aligned} h`nil &= nil \\ h`(a, nil) &= f`a \\ h`(a,(b,c)) &= g`((((a,b),c), h`(a,b)), h`(a,c)) \end{aligned}$$

Finally, equality on functions is defined extensionally.

- There is the sort C of classes. We are given the class containing only nil, i.e. $\{nil\}$ and have binary intersection and union $\cup$ and $\cap$, as well as the inverse image of a class c under a function f, $f^{-1}c$ where $t \in f^{-1}c \leftrightarrow f`t \in c$.
  More complicated, we can also build recursively enumerable classes $Ic(a, b)$, which is the class containing a and closed under the rule $t_1, t_2/t$, where $((t, t_1), t_2) \in b$. Equality, and the subset relation, on classes are defined extensionally.
- Finally, we have induction. Over the universe,

  $$nil \in c \to \forall a, b(a, b \in c \to (a, b) \in c) \to \forall x(x \in c)$$

  and over inductively defined sets,

  $$c' \subset c \to \forall a, b, c(b, c \in c \to ((a, b), c) \in c'' \to a \in c) \to Ic(c', c'') \subset c.$$

FS_0 is intended for building Gödel-encoding of formal languages and theories, and this is done by building classes that define well formed, or provable, formulae; e.g. we could define the class of all well-formed formulae of first order logic (encoded as s-expressions). When we try to do this, however, it soon becomes clear that enormous and painstaking effort is needed to build these by hand and at the end the definitions are not are not intuitive; further, when we try to prove that what we have produced has the properties that we want, we find, almost invariably, that it is full of hard to correct errors. Further, there is no way to structure developments very effectively.

We do, however, have a theorem that characterises which classes we are able, in theory, to define, in the form of a comprehension principle. Given the definition

Definition 1 *We define the class of Σ^0_1-formulae to be the class containing equalities and inequalities between S sorted terms, and set membership, and closed under disjunction, conjunction and existential quantification of S sorted variables.*

then we have a comprehension theorem,

Theorem 1 (Feferman) *Given a Σ^0_1-formula $P[x]$ with one free variable x of type S, there exists a class c, such that $FS_0 \vdash x \in c \leftrightarrow P[x]$.*

Σ^0_1-formulae provide an expressive specification language: with them we can define any recursively enumerable set of s-expressions (which includes sets of provable formulae, of course). But this comprehension result is a meta-theorem; it is neither a schematic axiom nor a theorem of FS_0, and while we could add it as an axiom to the theory directly, that is precisely the sort of extension that we want to avoid. How we provide comprehension is in fact one of the main facilities that we document.

§3 Isabelle

The Isabelle pds [5] comes as a collection of extensions for an SML programming system. It cannot accurately be described as a program that just happens to be written in SML; the relationship between the two is much closer than that. We work with Isabelle directly through the SML command line, meaning that we also have direct access to SML to program extensions, or as a tactic language; a powerful but safe facility—the strong typing acting as an effective prophylaxis against accidental, unnoticed damage being done to an implemented theory.

The system is based on the observation dating back to the Automath [2] project, that a good foundation for a generic pds is type-theory, or typed lambda calculus, since it is possible to encode the proof systems of many logics (particularly those that can be reasonably expressed in a natural deduction style) directly in the $\forall, \rightarrow$ fragment of such a theory and that in doing this we finesse the traditionally ubiquitous problems with variable binding or capture, and substitution.[2] For details of the type theory provided by Isabelle see [5]. We can think of Isabelle as a collection of tools for deriving and manipulating terms in type theory.

It is important to stress several unusual points about Isabelle. The terms that are derived in it are terms in a typed lambda calculus, some of which happen to encode terms in a declared logic, but these are not the only terms Isabelle works with: other terms represent rules in declared logic, and these can be derived too. In fact perhaps the best way to think of Isabelle is as basically a system for deriving rules rather than theorems; theorems are simply a degenerate case of rules with no premises and no schematic variables. Also, Isabelle is not, based on rewriting; lambda terms (i.e. encoded rules) are combined together rather by (higher order) unification and proofs are thus built by resolving rules represented as implicational terms in the type-theory against forumalae to be proved (also terms in type theory). This has some interesting effects; for instance we can use the same rule for both forward and backwards proof (since unification is 'bidirectional'), and it is possible to have metavariables in formulae that are to be proven, which can be instantiated in the course of the proof, picking up information from rules as they are used.

The details of how Isabelle allows access to the type theory through SML are as follows:

- A new data-type `theory` for packaging the collections of constant definitions that make up a declared theory.
- Functions for combining and extending `theory`s; i.e. if we have defined a `theory` encoding first-order logic and call it `Pred`, we can extend it with definitions for natural numbers to generate a new theory which we might call `Nat`, or combine `Nat` with, say, `List` to generate a theory of naturals and lists which inherits all the theorems that have been proven for either.
- A new data-type `thm` for the axioms of the `theory`s we have defined, and also for the terms we have derived in the 'theory's we have defined. In fact, and importantly, in Isabelle there is no distinction between derived and basic theorems of a theory, they are all just `thm`s.

Along with the the basic system, we also get some tools for building things like rewrite systems, and a few predefined logics, including sorted classical first-order logic.

Since classical first order logic is already available, we can immediately define, as an extension of it, a basic system for FS_0, all we have to do is declare sorts S, F, C then we can type in the axioms literally (allowing for the restrictions that a typewriter imposes compared to a typesetting system) as we find them in [3]. And we have a naïve implementation of FS_0.

§4 The basic implementation

As we have already said, a naïve implementation of FS_0 is not usable, but we can take stock of what it does make available (in Isabelle).

What this mostly amounts to is an effective method for modularising development. We have said that in Isabelle we can define a theory as an extension of another; thus the idea of a theory simply as a definitional extension of another is very natural: we simply add a new constant, and make it equivalent to the formula or term that it is abbreviating in the new extended theory.

Now, since a theory in FS_0 is basically a collection of FS_0 terms, we can take over this facility for abbreviation as definitional extension in a new theory and use it

[2] Of course, FS_0 is in a completely different tradition to this, and we have to build our own binding mechanism, but is intended for different purposes—we do find it pleasing that one framework logic should be so effective for implementing another.

for defining new FS_0 theories, each of which *is* a new Isabelle theory, albeit only a definitional extension. For a collection of classes, functions and s-expressions that we want to define as a module, we make a definitional extension of some earlier theory (which might be root FS_0, or itself some extension) and package these together as a new theory. Then we can prove the basic theorems that show that the definitions have the properties we want. From then on, all the the messy details of the implementation work can be hidden behind the abbreviations for the definitions, and theorems about them. And since we can combine these theories in Isabelle, with the result inheriting the theorems of its ancestors, we have a simple but effective tool for structuring the development of large theories as collections of small ones.

There are two ways to define a new theory as an extension of an old. The the first, 'basic' way is to use the `extend_theory` function that comes with the system. This is messy, since it takes a large and complicated collection of half a dozen or more arguments. The second is to use the Isabelle front end; this is a preprocessor that takes files which contain the equivalent of the information needed by `extend_theory`, but in a much more readable format, generates the theory and packages it, along with the various new axioms and other declarations that have been added, as an SML structure. Unfortunately neither of these methods is really suitable. The second method is not suitable because to use it we need to type the details of terms in by hand, and we have already explained why we want to avoid that). The first is unsuitable not only because here too, we would need to type the terms in by hand, but because it is simply too complicated (since we are only interested in definitional extensions).

§4.1 *What do we need?* At this point we have to consider in more detail exactly what we need to be able to do in an implementation. We automatically have a way to structure theories declared in FS_0, but anything else has to be built. And we know that we want to avoid having to construct, by hand, large terms to be assigned to abbreviations.

If a collection of definitions is constructed by hand (as happened in [4]) then the first thing that has to be done is to prove a collection of theorems describing (and checking) what those definitions actually do; i.e. translating them back into logical propositions. For instance we know from the comprehension theorem (theorem 1) that there is a close relationship between FS_0 classes and a certain class of logical formulae, and we have explained that this relationship is central to how we use the theory, so, if possible, we would like direct access to it. But as well as classes, we also want to define new functions, where there is no clear relationship like for classes. However, the idea that allows us to provide comprehension can be generalised (if not so elegantly) to provide a mechanism for generating terms from statements of their extensional properties.

The facilities for constructing classes and functions that we have implemented are very effective for connecting a defined object to a formula giving its properties. However they are also 'bottom up', since they build objects out of basic components. This is very safe, or course, since it ensures that everything we define *is* a definitional extension of FS_0. This means that the 'top down' approach to development is not possible. Thus we also provide a way to add, temporarily, new constants to the theory, along with new axioms, instead of just definitional extensions perhaps so that development on it can be postponed, or maybe continued in parallel.

We shall describe each of these in turn in the next section.

§5 Building definitions

Thus we have defined a new function `extend_FS0_theory` (by analogy with `extend_theory`) which takes the theory to be extended and a list of definitions and returns a package of a new theory and an association list of useful theorems that we have been able to generate automatically. Currently four sorts of definitions can be put on this list, and we discuss them one at a time.

§5.1 *Simple constants* The first sort of definition allows the definition of simple constants where we can type the body of the definition in whole.

For instance, in a theory of natural numbers, the natural numbers can be modeled as lists of *nil*s; i.e. $0 \rightsquigarrow nil$, $s(0) \rightsquigarrow (nil, nil)$, $s(s(0)) \rightsquigarrow (nil, (nil, nil))$ etc. Then on the list of definitions would be the declarations

```
def("zero", "nil"),
def("s", "[const(nil),Id]")
```

which results in the new theory containing the new name *zero* for *nil* and the new function s, where $s`t = (nil, t)$. Note that this last fact is a theorem that we have to prove, `def` declarations are so general that it is not possible to extract any extra information from them; however we can see the beginning of an abstract interface here: we can try to provide a set of theorems that talk about the abstract behaviour of s and *zero* in the natural numbers, without regard to their implementation.

§5.2 *Comprehension* A `def` declaration is not really different from the sort of declarations that the standard Isabelle front end can handle. More interesting is comprehension. We would like to make this available in some convenient way. The secret to doing this is to examine the proof (on paper) of the comprehension theorem. This shows, by induction on the structure of Σ^0_1-formulae, that it is always possible to construct a suitable class. Most of the cases are easy; for instance there is an obvious equivalence between $\cap, \cup$, and $\wedge, \vee$. The most tricky case is for $\exists$, were we have to show that given $x \in c \leftrightarrow P[\pi_1`x, \pi_2`x]$, then there is some construction $Ex(c)$ such that $x \in Ex[c]$ iff $\exists y.P[x, y]$. The construction of Ex is a bit tedious but not impossible (see [3] for the details).

Unfortunately the type theory of Isabelle is too weak to formalise all of this argument, since it does not support induction. The induction is the only thing that cannot be formalised though; all the step cases are provable; i.e. we can derive rules for each possible reduction step needed by the proof. Then, since rules in Isabelle are implicational formulae in higher order logic, and proofs are built by resolving those rule against the formula to be proven, the equivalent of the induction can be implemented as a simple backchaining algorithm (which is, in fact, deterministic, since there is exactly one rule that resolves against each case in the definition of Σ^0_1-formulae.

Thus for existential quantifiers, we can prove the rule

$$\frac{\forall x(x \in z \leftrightarrow P[\pi_1`x, \pi_2`x])}{\forall x(x \in Ex[z] \leftrightarrow \exists a(P[a, x]))}$$

(given some z such that..., then there is some z' (actually $Ex[z]$) such that ...).

Thus, if we wanted to define the class of the 'less than' relation, we could start with a goal of the form

$$\forall x(x \in ?\mathbf{c} \leftrightarrow \exists w, y, z(x = (y, z) \wedge plus`(x, w) = z))$$

(where $?\mathbf{c}$ is a metavariable hole in the goal). We can immediately apply the rule for the existential case, which reduces the goal to

$$\forall x(x \in ?\mathbf{d} \leftrightarrow \exists y, z(\pi_2`x = (y, z) \wedge plus`(\pi_2'x, \pi_1`x) = z))$$

now $?\mathbf{c}$ has been instantiated with $Ex[?\mathbf{d}]$. We can repeat this step twice more, then we change to the rule for $\wedge$ and so on. Eventually we have only goals of the form

$$\forall x(x \in ?\mathbf{e} \leftrightarrow x \in d)$$

which can be made true by unifying with

$$\forall x(x \in y \leftrightarrow x \in y)$$

which sets $?\mathbf{e}$ equal to d, then we can look at $?\mathbf{c}$ to see what it has been instantiated with, which gives us the class term we are looking for. This way, not only do we generate the class term that we want from the property we want it to satisfy, but we get, for free a theorem that states that it satisfies that property.[3] This approach is similar to the idea that Basin suggests in [1], as a general method for program synthesis.

We have implemented this so that we can write a definition, directly, as

```
comp("ltC", "(y,z)", "EX w. plus`(y,w)=z")
```

(i.e. the class of all instances of the term `(y,z)` such that...) Then the whole procedure just described is performed automatically: first an equivalent term, with just one free variable, is constructed, which has the form

```
x:?c <-> EX y z w. x = (y,z) & plus`(y,w)=z
```

then this is set as the goal to be proven and the proof procedure we have just described is run on it. The class is generated and attached to a name, then a version of the theorem defining the comprehension relation, only with the new name substituted for the generated class term, is proven and returned.

Thus we have solved both problems at once: the constant defining the class has been generated, automatically from a clear specification, and at the same time, a theorem connecting the specification and the class together by a logical equivalence has been proved, so it should never, in future, be necessary to unfold this class definition to get at what is inside it.

§5.3 *general synthesis* We have described a powerful method for building classes in FS_0 as conservative extensions, but general though it is, it is not always suitable, and anyway we also want to define, similarly, new functions as conservative extensions; and unfortunately no similarly elegant solution is available for that.

However the situation can be improved far beyond having to piece functions together by hand out of primitives, and then proving that the result has the right properties. In fact, we can extend the idea that we have just used to implement the comprehension theorem to a much larger class of synthesis problems.

Above, we have a uniform procedure which when given theorem-with-a-hole of a particular form, can fill out that hole. But in general no such uniform procedure is available; instead, a custom proof has to be provided. As an example, consider the 'less than' ordering again, only this time we want to define it as a function, not as a relation. We can adopt the same method to synthesise it, starting with a formula-with-a-hole that we try to prove, as follows:

$$?\mathbf{lt}`nil = false \wedge ?\mathbf{lt}`(a, zero) = false \wedge \\ ?\mathbf{lt}`(a, s`a) = true \wedge (a \neq b \rightarrow ?\mathbf{lt}`(a, s`b) = ?\mathbf{lt}`(a, b))$$

The $?\mathbf{lt}$ here can be filled in in exactly the same way as $?\mathbf{c}$ above, resulting in a definition that can be read off, and theorem giving properties of that definition. The difference here is that we have to find a proof ourselves.

This can still be partially automated though. We just have to arrange to tell the system somehow what the right way to go about proving this goal, apart from the results are the same. Thus we find the entry

```
sch_def("lt",
   "?lt`nil=true & ?lt`(a,zero)=false &
    ?lt`(a,s`a)=true) & (a~=b --> ?lt`(a,s`b)=?lt`(a,b))"
   ltsynthtac)
```

where the extra argument, `ltsynthtac`, is a program to prove the goal, that replaces the uniform proof procedure for comprehension. Assuming that `ltsynthtac` doesn't fail, the result of running it is exactly like before: the new function is synthesised and attached to a new constant, and the theorem that has been proved is returned, with the new abbreviation replacing the synthesised term.

[3] Essentially the induction which was not possible in Isabelle has been added informally, using SML.

In fact, we could have defined `ltC` in the same way, by giving the definition

```
sch_def("ltC",
        "x:?ltC <-> EX y z w. x = (y,z) & plus`(y,w)=z",
        comprehension_tac)
```

As we said, this is clearly not quite as elegant as that for comprehension, for all sorts of reasons: it requires a user to build the tactic that is to be given as a parameter, which can be quite tricky, and it does not guarantee that the theorem returned is an exact and complete description of the object that has been synthesised, but it is, nonetheless, very effective, and, of course, very general.

§5.4 *top down specification* It is part of the received wisdom that large systems should, at least in part, be developed 'top down'; i.e. the implementation should be developed by repeatedly refining the abstract definition into somethingc oncrete. This, it is argued, helps to keep the development under intellectual control. We cannot do this with the facilities we have defined so far: anything we define has be be built from the ground up. We add a fourth sort of entry on the range of possible definitions, to allow a top down style. This looks very like `sch_def`, but has one less parameter.

```
abstract("lt",
   "lt`nil=true & lt`(a,zero)=false &
    lt`(a,s`a)=true) & (a~=b --> lt`(a,s`b)=lt`(a,b))")
```

(notice that the formula has no holes) With `comp` and `sch_def` a definitional extension of the given theory is generated and a theorem about it is proven. With `abstract` this process is short-circuited: no effort is made to try to generate a suitable definition, or prove a theorem. The new theory is extended with a new constant `lt`, and a new *axiom* defining its behaviour. This is dangerous because this extension is not definitional and the associated theorem is not a theorem of FS_0 (there is nothing to stop us adding a false axiom), but as was said earlier, `abstract` is supposed to be used only as a temporary, stop-gap, measure, and removed before the end.

§5.5 *Taking stock* If we take stock of what we have done in this section we see a single idea, presented in a variety of ways. We have tried constantly to hide the details of FS_0 definitions behind abbreviations, which we treat as new objects, with new defining axioms, added to the theory. FS_0 is, in fact really being used only as an underlying foundation to the extension we define. However, if we really need to, we can, at any time, strip these levels of abstraction away, leaving the unadorned theory, since everything is, in the end, just a definitional extension. We will extend this theme in the next section, when we talk about how Isabelle allows us to implement rewriting.

§6 In use

We have described a modification of Isabelle that we have found to be a practical way of building theories as definitional extensions of FS_0. We now show that it is a practical way of working in those theories. There are several aspects to this work, which we will discuss in turn.

§6.1 *Induction* As we have said earlier, a lot of the work involved in using FS_0, is building classes. This work is more ubiquitous even than we have implied, since the typical method of proof in the theory is induction, of one sort or anther, and induction is only available over classes, in spite of the fact that we almost never want actually to do this. Again, as a result of the comprehension theorem we know that there is an equivalent class for any Σ^0_1 predicate, and thus we have induction over this class of formulae (which is enough in for most practical things). The metatheory of Isabelle will not allow us to prove derived rules of this form; e.g. we cannot prove as a single metatheorem that

$$\frac{P[nil] \quad \forall x, y(P[x] \to P[y] \to P[(x,y)])}{\forall x P[x]}$$

(where P is a Σ^0_1-formula). We are, however able to prove something almost identical in the rule:

$$\frac{\exists c \forall x(x \in c \leftrightarrow P[x]) \quad P[nil] \quad \forall x, y(P[x] \to P[y] \to P[(x,y)])}{\forall x P[x]}$$

(no side condition). This produces an extra goal, of course, which corresponds to the side condition on the previous rule since we know that it is exactly these classes that is defineable. And the extra goal is not a problem, since it can be disposed of immediately and automatically with the same tactic that we use to implement generation by comprehension in `comp`.

§6.2 *Rewriting* The other large and ubiquitous problem-in-use for FS_0 is term simplification, since many theorems are proven mostly by selecting a suitable induction then simplifying the resulting terms. But this work of term simplification is tedious and difficult to do by hand.

It maybe not immediately clear why this is a problem. The functions we are able to define have a well defined structure with obvious rewrites (described in 2); surely we need only implement this, and arrange for abbreviations to be unfolded as necessary.

But consider a simple example: we need a function to number the elements of a list with their positions. We can specify this as follows:

$$\begin{aligned} label`l &= labeld`(zero, l) \\ labeld`(n, nil) &= nil \\ labeld`(n, (f, r)) &= ((n, f), labeld`(s`n, r)) \end{aligned}$$

and it is not hard, though tedious, to define first-order primitive recursive (i.e. using *Rec*) versions of *label* and *labeld* to satisfy this specification. But if we try to evaluate the resulting program using the strategy we have just described, we discover that the path the evaluation takes looks nothing like the specification suggests it should be. If the term to be normalised is ground, we get the result we want, but as 'raw s-expressions'; that is, many of the abbreviations in subterms will have been unfolded (and without these abbreviations, the term is an incomprehensible s-expression built of nothing but *nil*).[4] If the term is not ground, the results will be much worse. In fact the naïve term reduction strategy suggested by the axioms is useless.

What is needed is a rewriting system that respects the specifications, and abstract properties of what has been defined, not the concrete details of the implementation. However, Isabelle, provides a general rewrite package (`term_simp_tac`) that can be effectively used for our purpose. This takes, as a parameter, a set of equational theorems that are to be used as rewrite rules. These theorems need not describe the actual normalisation path of a set of terms (the package accept a diverging set of rules); they must only be provably correct. Thus once we have verified that *label* and *labeld* behave as they should, according to their specification above (we need induction for this) we can use the equations that specify their behaviour as their rewrite rules. Such equations are always available, since whenever a function is defined, the first thing we have to do is check that it does what it should. In fact if, like in the case of `lt` above, which is defined with a `sch_def`, we get the desirable rewrite properties at the same time as we synthesise it.

[4]It worth mentioning that in this case even the technique that has been used in some systems, where unfolded abbreviations are tracked, so to be folded back afterwards when possible, does not work.

Thus we have continued the the theme of abstraction that we started in the last section. There we provided facilities for defining objects in abstract terms, trying as much as possible to hide the underlying structures used to build them. Here we bring that process to a conclusion by implementing rewrite for the functions we define in those same abstract terms, again hiding (and thus allowing a user in future to ignore) the underlying structure. We are thus able to provide efficient, and completely abstract interfaces for the various theories that we build, and we can combine them using just these interfaces.

§7 Development experience

We have described the basic facilities that we have implemented for working with FS_0. We now describe some of the development that we have actually carried out.

FS_0 is designed for doing proof theory, and therefor for formalising mathematical theories. But most theories have some notion of binding, and substitution, and (unlike type theoretic logics for encoding) it does not have these, so they have to be developed inside the theory. However, with a sufficiently general idea of what sort of facilities are needed, the work of implementing this should be reusable for any theory. I.e. rather than define binding for first-order logic, the lambda calculus, and higher order logic all separately, we could define the term structure of each of these on top of some more abstract theory, and this is what we have done. The largest theory we have implemented is for the 'binding structures' proposed by Talcott [6]. However this is a large and complicated theory (the implementation details will appear in another paper) so the development has been structured. In fact we have developed in all the following theories: natural numbers, lists, finite functions (from lists), and a general system of binding structures. However the intention of this paper has been to concentrate on the implementation of FS_0 that we have built, rather than to report on the details of how theories are developed in it.

§8 Conclusions

At the centre of any formal proof development system are two things: an underlying logic which can be used to build various theories, and a way to make that work of building as easy as possible, by allowing users to impose various sorts of structure on the development. This latter might provide (among other things)

- Modularity: it should be possible to develop parts of a theory as separate chunks, which can be combined as necessary.
- Top down development: it should be possible to assume that certain theories are available, even if the supporting development has not been finished, allowing that effort to be postponed or performed concurrently.
- Abstraction: it should be possible to present an interface to a theory that hides the (possibly messy) details of the implementation behind an abstract description of its behaviour.

We have shown that in a system like Isabelle, which provides sophisticated structuring and development facilities of the sort we have listed, that are provided prior to, and therefore independent of, any theory we might implement, we can take a simple (= primitive) theory, in this case FS_0, lacking any such facilities and provide them, while preserving the simplicity of the theory. We have been able to provide a very practical and usable system which is implemented using exactly the axioms that we find in Feferman's original paper; all development in the system can easily be unwound to that level.

We even believe it would be difficult to build a custom theorem prover as effective as what we have produced, since many of the more useful facilities are inspired directly by what Isabelle provides, and these facilities are so much more powerful than we can imagine trying to program 'from scratch'.

System availability Parties interested in getting a copy of the code for the implementation should send e-mail to the address given at the beginning of this paper); we hope to have it ready for release in the immediate future.

Appendix: !An example development

In this appendix we list some code, and some sample development from a theory (the of lists).

The Isabelle development of a theory can be packaged inside a structure, which means that it can present a fairly abstract interface, which is of the following form:

```
signature LISTS =
    sig val thy : theory
        val thms : (string * thm) list
        val list_ax1 : thm
        val list_ax2 : thm
        val list_ax3 : thm
        val ListRec1 : thm
        val ListRec2 : thm
        val ListRecTyping : thm
        val inject1 : thm
        val inject2 : thm
        val injectTyping : thm
        val map1 : thm
        val map2 : thm
        val mapTyping : thm
        val ListIndg : thm
        val List_ss :simpset
    end;
```

i.e., the rest of the system should see the development as consisting of a theory `thy`, an association list `thms` which contains a bunch of information about comprehension generated by the comprehension tactics, that we do not want to use explicitly (and could not really, even if we wanted to) and a list of theorems which together (should) present to the rest of the system an abstract view of the concrete structures that we have developed. Unfortunately the typing of ML is not enough to make the details of a `thm` explicit (i.e. what exactly it is a theorem about), but only enough to ensure that it *is* a theorem of some sort. Thus the typing details of signature cannot give all the details that we would like. Then, inside a `lists:LISTS=struct`, `end` pair we can build concrete objects in FS_0, prove theorems about them, and then assign some of these theorems to the names listed in the signature.

First we build the basic theory, which is a collection of constant definitions, as follows:

```
val (thy,thms) =
  extend_FS0_theory basics.thy "sorted lists" []
      [def("empty_List", [], "nil"),
       comp("base_List", [], "a", "a=empty_List"),
       comp("step_List", ["sort"], "re2((h,t),t)", "h:sort"),
       def("List", ["sort"], "I2(base_List, step_List(sort))"),
       sch_def("ListF", ["f"],
         "?ListF=Rec(Id,andF * [f*arg1,rc2]) * [trueF, Id] &\
         \?ListF`nil=true & ?ListF`(a,b)=andF`(f`a,?ListF`b)",
         (fn _ => fn _ =>
            [...])),
```

```
sch_def("ListRec", ["base", "step"],
  "?ListRec=Rec(base, step * [[[arg0,arg1],arg2],rc2]) &\
  \?ListRec`(a,empty_List)=base`a &\
  \?ListRec`(a,((b,c)))=step`(((a,b),c),?ListRec`(a,c))",
  (fn thy => fn _ =>
      [...])),
comp("step_List_x", ["sort"],
       "re2((h,t),t)", "h:sort & t:List(sort)"),

def("Listarg0", [], "P1 * P1 * P1"),
def("Listarg1", [], "P2 * P1 * P1"),
def("Listarg2", [], "P2 * P1"),
def("Listrc", [], "P2"),

def("inject", ["f"],
       "ListRec(P2, f *\
       \[[P1 * Listarg0, Listarg1], Listrc])"),
def("map", ["f"],
       "inject([f*P1,P2]) * [[P1, const(empty_List)],P2]")
];
```

This constucts a new theory (with the name `"sorted lists"` as an extension of the earlier `basics.thy` and assigns it to `thy` while at the same time assigning details about how the various classes generated by the comprehension tactic correspond to the given first-order predicates, to `thms`.

Then, using these two, we are able to start developing the theorems that give the abstract properties of the objects we have just defined. For instance we can write

```
val List_ax1 =
    prove_goal thy
        "empty_List : List(D)"
        (fn _ =>[...]);
```

which proves that `empty_List` is a member of the class of lists generated from any class `D` (the details of the tactic used to prove it have been replaced with an ellipsis ...).

Or, more complex, we can prove one of the axioms that gives the properties of the `inject` combinator.

```
val inject2 =
    prove_goal thy
        "inject(f)`((k,b),(h,t))=f`((k,h),inject(f)`((k,b),t))"
        (fn _ => [...])
```

Then finally we have the theorem for induction over lists,

```
val ListIndg =
    prove_goal thy
        "[| x:List(C); !! x. f`x = true <-> x:C;
            !! x. x:X <-> P(x); P(empty_List);
            !! h s. [| h:C ; s: List(C); P(s)|] ==> P((h,s))
         |]==> P(x)"
        (fn [h1, h2, h3, h4, h5] =>
            [...]);
```

and we finish by building a suitable rewriting system `List_ss`, which rewrites in terms of these theorems (though it, of course, *doesn't* actually have to use them, or work through them).

Now, if we have chose our theorems well, we have a complete theory of lists which as far as the rest of the system is concerned, has been effectively abstracted away from impementational details. We are able to treat the set of theorems provided by the structure `list` as though they were simply axioms. The isolation is not quite complete, we can always go around this abstract interface, but there should not be much temptation to do so if the theorems are well chosen, since it is hard and messy to do. (If we wanted to be absolutely sure about what we have done, we could, of course, check for implementational bias).

References

1. D. Basin. Logic frameworks for logic programs. In *4th International Workshop on Logic Program Synthesis and Transformation, (LOPSTR'94)*, Pisa, Italy, June 1994. Springer-Verlag. To Appear.
2. 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.
3. S. Feferman. Finitary inductive systems. In D. Gabbay, editor, *What is a Logical System?*, Oxford, 1994. Oxford University Press. (also appeared in Logic Colloquium '88).
4. S. Matthews. *Metatheoretic and Reflexive Reasoning in Mechanical Theorem Proving.* PhD thesis, University of Edinburgh, 1992.
5. l. C. Paulson. *Isabelle: A generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, Berlin, 1994.
6. C. Talcott. A theory of binding structures, and applications to rewriting. *Theoret. Comput. Sci.*, 112:99–143, 1993.