STRUCTURED THEORIES IN LCF

D.T. Sannella and R.M. Burstall

Department of Computer Science
University of Edinburgh

Abstract: An extension to the Edinburgh LCF interactive theorem-proving system is described which provides new ways of constructing theories, drawing upon ideas from the Clear specification language. A new theory can be built from an existing theory in two new ways: by *renaming* its types and constants, or by *abstraction* (forgetting some types and constants and perhaps renaming the rest). A way of providing *parameterised* theories is described. These theory-building operations -- together with operations for forming a *primitive* theory and for taking the *union* of theories -- allow large theories to be built in a flexible and well-structured fashion. Inference rules and strategies for proof in structured theories are also discussed.

1 Introduction

Edinburgh LCF [GMW 79] is a mechanised system for conducting proofs interactively. Users prove theorems in LCF by writing (and then running) goal-directed proof strategies as programs in a general-purpose applicative language (ML). Although users are given complete freedom to try any proof strategy they choose (including an incorrect one) it is impossible to prove an invalid theorem in LCF. The system provides a set of primitive building-blocks which are useful for building proof strategies, but users are not compelled to make use of them.

The proof of a theorem takes place in the context of a *theory* -- that is, of some set of types (type operators, since polymorphic types are allowed), constants and axioms forming the axiomatisation of some problem area. New theories can be built by combining several existing theories and enriching the result with some new type operators, constants and axioms. A hierarchy of theories can be built in this fashion.

We propose an extension to LCF whereby theories can be built in new ways. Most of this has been implemented but is not as yet much used. The type operators and constants of a theory may be *renamed* to produce a new theory, or we can *abstract* from a theory (forgetting some of the type operators and constants and perhaps renaming the rest). We also describe a way of providing *parameterised* theories. These theory-building operations -- together with operations for taking the *union* of two theories and for forming a *primitive* theory -- allow large theories to be built in a flexible and well-structured fashion. Such a *structured* theory inherits the type operators and constants of the theories from which it was built (after appropriate renaming) but does not directly inherit the axioms.

Proving theorems in a structured theory is different from proof in a conventional LCF theory. In the course of a proof, the user will change frequently from the context of one theory to that of another, climbing down the hierarchy to prove simple lemmas in basic theories and then up again to apply these lemmas in the proof of theorems in higher-level theories. We provide some new inference rules to permit such a style of proof, and we develop a general strategy for proof in a structured theory. The structure of the theory can be useful in directing the search for a proof. In many cases the problem of gathering together enough information in a well-structured theory to prove a theorem can be solved mechanically.

We have been strongly influenced by work in structured algebraic specification (e.g. [ADJ 78], [ASM 79] and [Bau 81]) and in particular by our work with the Clear specification language [BG 77]. The semantics of Clear can be expressed in terms of the structuring operators mentioned above. Our experience with Clear has convinced us that it is important to retain the structure which is formed as a large theory is built up in stages. Structuring is necessary to keep the information in large theories under control. This is our attempt to transport ideas from Clear to the LCF context.

2 Edinburgh LCF

We now briefly describe the features of Edinburgh LCF which are most important for our purposes. A full description of the system is given in [GMW 79].

Edinburgh LCF (sometimes called simply 'LCF') is a system for conducting proofs interactively. It can be viewed as consisting of three relatively independent components: ML, the metalanguage in which proofs are carried out; PPLAMBDA, the underlying deductive calculus; and a methodology for goal-directed proof in PPLAMBDA using ML.

ML is a general-purpose applicative language incorporating a completely secure higher-order and polymorphic type discipline. It includes a flexible mechanism for raising and trapping exceptions and allows the declaration and use of abstract data types which are accessible only through the functions provided when the type is declared. Ordinary types used in programming (such as integer, boolean and list) are predefined in ML as well as special types (like term, formula and theorem) for use in proving theorems.

PPLAMBDA is a family of deductive calculi or theories with terms from the typed lambda-calculus and (for each member of the family) a set of type operators, constants and axioms. PPLAMBDA can be viewed as a collection of ML functions and types. However, for reasons of efficiency it is in fact implemented only partially in ML. A *theorem* (thm) in PPLAMBDA is an ML data structure like a term or formula, but with a crucial difference: the only way to construct a theorem is by application of built-in inference rules (i.e., thm is an abstract data type with inference rules as constructor functions). This ensures that any object of type thm must be true in the theory in which it was formed. Thus the type security provided by the ML type checker is used to maintain logical security. There are facilities for building a new PPLAMBDA theory by combining several theories and enriching the result with some new type operators, constants and axioms. This allows a group of theories to be structured into a hierarchy in which a theory inherits all the type operators, constants and theorems (provable sentences, including axioms) of the theories from which it was built.

Given a theorem to be proved (we use the notation $A \overset{?}{\vdash} c$, where A is a list of assumptions and c is the desired conclusion), we apply a *tactic*; that is, a proof rule in the form of an ML function of type $goal \rightarrow (goal^* \times (thm^* \rightarrow thm))$. This may fail if the goal is not of the appropriate form. If it succeeds then it delivers a list of subgoals together with a *proof*; this is a function built from inference rules which will produce a theorem (written $A \vdash c$) corresponding to the original goal if it is given a theorem corresponding to each of the subgoals. Proving a theorem is then a matter of applying one tactic after another until the empty list of goals is obtained. The system supplies a collection of built-in tactics, but the user may construct his own. *Tacticals* like

THEN: tactic $\times$ tactic $\rightarrow$ tactic

are provided for composing tactics into larger tactics called *strategies*. Typically, a user proves a theorem by interactively designing a strategy which will solve the entire problem by reducing his goal to the empty goal list.

3 Structured theories

Edinburgh LCF as described in the previous section is a powerful tool for interactive proof. This has been demonstrated by the success of a number of attempts at applying LCF to prove rather difficult theorems -- see for example [Cohn 79]. But one weakness of LCF is that only a primitive facility is provided for structuring theories. Using the means described earlier, structures such as the following can be built:

$$\begin{tikzcd}
 & & & T & \\
 & & T3 \arrow[ur, hook] & T4 \arrow[u, hook] & \text{extension C} \arrow[ul, hook] \\
 & T1 \arrow[ur, hook] & & T2 \arrow[ul, hook] & \\
\text{extension A} \arrow[ur, hook] & & T0 \arrow[ul, hook] \arrow[ur, hook] & & \text{extension B} \arrow[ul, hook]
\end{tikzcd}$$

In this diagram, extension C denotes the new type operators, constants and axioms by which the union of T3 and T4 is enriched to yield T. Extension C is therefore itself a theory. Each theory inherits the type operators, constants and theorems of its parents (where the parents of T are T3, T4 and extension C, for example). Separate theories may share common ancestors. The theory containing the inference rules of PPLAMBDA is implicitly a parent of every theory.

The problem is that we sometimes would like to build theories in other ways. For example, suppose we build the theory of lists from the theory of booleans:

<u>List</u> = the extension of Boolean by:

types	list of α	
constants	nil: list of α	
	cons: α X list of α $\rightarrow$ list of α	
	head: list of α $\rightarrow$ α	
	tail: list of α $\rightarrow$ list of α	
	null: list of α $\rightarrow$ bool	
axioms	head(cons(x,l)) = x	null(cons(x,l)) = false
	tail(cons(x,l)) = l	null(nil) = true

(Free variables are implicitly universally quantified.) The theory of stacks may be built in the same way:

<u>Stack</u> = the extension of Boolean by:

types	stack of α	
constants	nilstack: stack of α	
	push: α X stack of α $\rightarrow$ stack of α	
	top: stack of α $\rightarrow$ α	
	pop: stack of α $\rightarrow$ stack of α	
	isempty: stack of α $\rightarrow$ bool	
axioms	top(push(x,s)) = x	isempty(push(x,s)) = false
	pop(push(x,s)) = s	isempty(nilstack) = true

Note that Stack is identical to List except for type and constant names. We would like some way of describing this relation within LCF.

Our solution is to extend LCF to allow new kinds of relationships between theories in addition to the present 'is an enrichment of' association. In this case instead of constructing Stack by extending Boolean we can build it from List by renaming the type operator *list* as *stack*, and the constants *nil*, *cons*, *head*, *tail* and *null* as *nilstack*, *push*, *top*, *pop* and *isempty* respectively (or List could be built from Stack by renaming in the opposite direction). If σ describes this renaming (details below) then we can form the following structure:

$$\begin{tikzcd}
& \text{Stack} & \\
& \text{List} \arrow[u, "\sigma"'] & \\
\text{list-extension} \arrow[ur, hook] & & \text{Boolean} \arrow[ul, hook]
\end{tikzcd}$$

Now although Stack has List and Boolean as ancestors, it inherits their type operators, constants and theorems only *after* they have been renamed according to σ. So Stack contains exactly the components given above, and not (for example) the theorem *tail(cons(x,l))* = *l*. An advantage of relating Stack to List in this fashion over building the two theories independently is that now any theorem proved about lists automatically extends to a corresponding theorem about stacks, without a separate proof.

As another example, a theory of symbol tables (for an Algol-like language with nested blocks) may be built from Stack and the theory of arrays (see [GHM 78] and [BG 77] for variations on this example):

```
Array  =  the extension of Index by:
types       array of α
constants   nilarray:  array of α
            put:  index  X  α  X  array of α  →  array of α
            get:  index  X  array of α  →  α
            isin:  index  X  array of α  →  bool
axioms      get(i,put(i,x,a))  =  x           not(i=j)  ⟹  get(i,put(j,x,a))  =  get(i,a)
            isin(i,put(i,x,a))  =  true       not(i=j)  ⟹  isin(i,put(j,x,a))  =  isin(i,a)
            isin(i,nilarray)  =  false
```

```
SymbolTable  =  the extension of Stack and Array by:
constants   addid:  index  X  α  X  stack of array of α  →  stack of array of α
            retrieve:  index  X  stack of array of α  →  α
            isinblock:  index  X  stack of array of α  →  bool
            enterblock, leaveblock:  stack of array of α  →  stack of array of α
axioms      addid(i,x,st)  =  push(put(i,x,top(st)),pop(st))
            not(isin(i,a))  ⟹  retrieve(i,push(a,st))  =  retrieve(i,st)
            isin(i,a)       ⟹  retrieve(i,push(a,st))  =  get(i,a)
            isinblock(i,st)  =  isin(i,top(st))
            enterblock(st)  =  push(nilarray,st)
            leaveblock(st)  =  pop(st)
```

So far we have built the following structure:

$$\begin{tikzcd}
& \text{SymbolTable} & & & \\
\text{symboltable-extension} \arrow[ur, hook] & \text{Stack} \arrow[u, hook] & & \text{Array} \arrow[ull, hook] & \\
& \text{List} \arrow[u, "\sigma"'] & & \text{Index}^* \arrow[u, hook] & \text{array-extension} \arrow[ul, hook] \\
\text{list-extension} \arrow[ur, hook] & & \text{Boolean}^* \arrow[ul, hook] \arrow[ur, hook] & & \text{index-extension}^* \arrow[ul, hook]
\end{tikzcd}$$

[The theories marked with * were not shown above.]

But now if we work in the theory SymbolTable we are forced to use the type *stack of array of α* when we mean *symboltable of α*, and we must use the constant *nilstack* instead of *niltable*. Moreover, since the theory SymbolTable has Stack and Array as parents, it inherits all of the type operators, constants and theorems of these theories. Many of these -- the type operators *stack* and *array* along with most of their associated constants and the axioms which define them -- are irrelevant to the new theory beyond the purpose they served in helping to describe symbol tables. We would like to *abstract away* from the particular construction we used to define symbol tables, retaining only the type operators and constants we need to use symbol tables and the theorems which define them.

Naturally, in the course of proving a theorem it may be necessary to refer to Stack and Array in order to determine the properties of symbol tables, but in the meantime they should not intrude.

We can build the theory we want from Symboltable by taking its *inverse image* under an appropriate renaming (this is equivalent to the **derive** operation in Clear). This renaming maps the type operators and constants we want in the result -- in this case these are the type operators and constants of Index and Boolean together with:

types symboltable of α
constants niltable: symboltable of α
addid: index $\times$ α $\times$ symboltable of α $\rightarrow$ symboltable of α
retrieve: index $\times$ symboltable of α $\rightarrow$ α
isinblock: index $\times$ symboltable of α $\rightarrow$ bool
enterblock, leaveblock: symboltable of α $\rightarrow$ symboltable of α

(but not *stack of* α or *array of* α or any of their constants) -- to the types and constants of SymbolTable. Here, *symboltable of* α maps to *stack of array of* α, *niltable* maps to *nilstack*, and the other type operators and constants map to themselves. Call this renaming σ'. The inverse image of SymbolTable under σ' contains a set of theorems formed using the type operators and constants shown above, consisting of just those sentences which σ' maps to theorems of SymbolTable. If we call the resulting theory BetterSymbolTable, then we have the following structure:

$$\begin{tikzcd}
 & & \text{BetterSymbolTable} \arrow[d, "\sigma'"] & & & \\
 & & \text{SymbolTable} & & & \\
\text{symboltable-extension} \arrow[urr, hook] & & \text{Stack} \arrow[u, hook] & & \text{Array} \arrow[ull, hook] & \\
 & & \text{List} \arrow[u, "\sigma"'] & & \text{Index} \arrow[u, hook] & \text{array-extension} \arrow[ul, hook] \\
 & \text{list-extension} \arrow[ur, hook] & & \text{Boolean} \arrow[ul, hook] \arrow[ur, hook] & & \text{index-extension} \arrow[ul, hook]
\end{tikzcd}$$

(An arrow pointing down denotes an application of the *inverse image* constructor. The arrow thus shows the direction of the renaming.) Again, although BetterSymbolTable has Stack and Array as ancestors, it does not inherit their type operators, constants and theorems. The relation between the theorems of BetterSymbolTable and the theorems of its ancestors is indirect -- to see if a sentence of BetterSymbolTable is a theorem, translate it using σ' and then try to prove the result in SymbolTable.

As the examples above show, we propose to change LCF to permit theories to be built in new ways from existing theories. We treat theories as ML data objects, and we build new theories from old theories by application of theory-constructing functions. Inheritance of type operators, constants and theorems from ancestor theories is indirect, even when a theory is constructed as in present-day LCF by combining two existing theories -- we believe that it is important to retain the structure which is formed as a large theory is built up in stages.

Proving a theorem in such a structured theory is different from proof in a conventional LCF theory. In ordinary LCF the user works within the theory he has chosen for the duration of his terminal session (although this theory may grow as he adds new type operators, constants and axioms). All of the theorems of the theory are immediately available for use in proofs, including the theorems of its ancestors. In contrast, the theorems of a structured theory tend to be scattered throughout the structure and must be extracted from the theories in which they reside when they are needed in a proof. In the course of a proof in a structured theory, the user may change from the context of one theory to that of another at will, climbing down the tree (more precisely, DAG) of theories to prove lemmas on basic types and then up again to apply these lemmas in the proof of theorems concerning higher-level types.

It may seem silly to distribute information in this fashion, in effect making it more difficult for a user to apply axioms and previously-proved theorems. But we argue that some scheme of this nature is necessary to keep the information in a large theory under control. Any sizeable unstructured assortment of theorems is more difficult to keep track of than the same collection of theorems organised into coherent theories, each containing only those theorems which are directly relevant to it. Moreover, as we will show later, the problem of finding all information relevant to satisfying a particular goal in a proof can be solved mechanically in a well-structured theory. A final reason for scattering theorems throughout a structured theory is that in the presence of the *inverse image* theory constructor, theorems cannot in general be brought up to 'top level'.

Although the LCF system was designed for conducting proofs in a particular logic (PPLAMBDA), much of the system including ML and the LCF proof methodology is logic-independent. In fact, David Schmidt at Edinburgh and (separately) Jacek Leszczylowski [Les 82] have done some work toward the development of an LCF system which will allow proofs to be conducted in any desired logic. The following formalisation of the theory-building operations is largely logic-independent as well. It does not depend on the particular inference rules or predefined type operators and constants of PPLAMBDA. Sentences need not be built from PPLAMBDA forms; any sort of sentence which is amenable to translation under a renaming is acceptable. See [GB 82] for the precise conditions which an acceptable logic (called an *institution*) must satisfy.

Def: A *signature* Σ is a pair $<S,\Omega>$ where S is a set of *type operators* (each having an *arity* $\in \mathbb{N}$) and Ω is a set of *constants* (each having a *type* constructed from operators in S and type variables).

The type operators and constants of each theory T form a signature, denoted *sig(T)*.

Def: A *signature morphism* σ: $<S,\Omega> \rightarrow <S',\Omega'>$ is a pair $<f,g>$ with $f:S\rightarrow S'$ an arity-preserving map on type operators and $g:\Omega\rightarrow\Omega'$ a type-preserving map on constants.

The 'renaming' σ:sig(List)$\rightarrow$sig(Stack) described above was a signature morphism. In particular, σ=<f,g> where:

f(*list*)=*stack*, f(*bool*)=*bool*
g(*nil*)=*nilstack*, g(*cons*)=*push*, g(*head*)=*top*, g(*tail*)=*pop*, g(*null*)=*isempty*,
g(*true*)=*true*, g(*false*)=*false*, g(*not*)=*not*, . . .

Now, σ is arity-preserving (because e.g. arity(*list*) = 1 = arity(f(*list*))) and type-preserving (because e.g. $f^{\#}$(type(*cons*)) = $f^{\#}$(αX*list of* $\alpha\rightarrow$*list of* α) = αX*stack of* $\alpha\rightarrow$*stack of* α = type(*push*) = type(g(*cons*)), where $f^{\#}$ is the extension of f to types). Note that a signature morphism need not be 1-1 or onto, although σ is both.

Def: If Σ=$<S,\Omega>$, then the *derived signature* dΣ is the signature $<dS,d\Omega>$, where dS is the set of types constructable from operators in S and type variables (the arity of a type is the number of distinct type variables it contains), and dΩ is the set of well-typed λ-expressions constructable from constants in Ω.

Def: A *derived signature morphism* $d\sigma:\Sigma\rightarrow\Sigma'$ is a signature morphism $d\sigma:\Sigma\rightarrow d\Sigma'$.

The renaming σ' described above was a derived signature morphism, σ':sig(BetterSymbolTable)$\rightarrow$ sig(SymbolTable). Indeed, more of the specification of BetterSymbolTable can be incorporated into this morphism. Suppose SymbolTable' is the same as SymbolTable above but without the constants *addid*, *isinblock*, *enterblock* and *leaveblock* (and without the axioms which define them). Let $d\sigma$:sig(BetterSymbolTable)$\rightarrow$sig(SymbolTable') be the derived signature morphism which is the same as σ' except that:

addid	maps to λi, x, st. push(put(i, x, top(st)), pop(st))
isinblock	maps to λi, st. isin(i, top(st))
enterblock	maps to λst. push(nilarray, st)
leaveblock	maps to pop

Then *inv-image(dσ, SymbolTable')* gives the same theory as *inv-image(σ', SymbolTable)* (= BetterSymbolTable), apart from structure.

If $\sigma: \Sigma \to \Sigma'$ is a signature morphism then let $\sigma^{\#}$: Σ-sentences $\to$ Σ'-sentences be the extension of σ to sentences.

Def: A *structured theory* is any term built using the following constructors:

prim-theory: signature X set of sentences $\to$ structured theory
union: structured theory X structured theory $\to$ structured theory
rename: signature morphism X structured theory $\to$ structured theory
inv-image: derived signature morphism X structured theory $\to$ structured theory

The semantics of structured theories is defined as follows:

sig: structured theory $\to$ signature

sig$[\![$ prim-theory(Σ, S) $]\!]$ = Σ if S is a set of Σ-sentences
sig$[\![$ union(T, T') $]\!]$ = sig$[\![$T$]\!]$ U sig$[\![$T'$]\!]$ if the signatures are compatible
sig$[\![$ rename(σ, T) $]\!]$ = Σ', where $\sigma: \Sigma \to \Sigma'$ if sig$[\![$T$]\!]$ = Σ
sig$[\![$ inv-image(dσ, T) $]\!]$ = Σ, where d$\sigma: \Sigma \to \Sigma'$ is a derived signature morphism if sig$[\![$T$]\!]$ = Σ'

Terms which fail to satisfy the indicated conditions above yield errors. Otherwise, the provable theorems of a structured theory are as follows:

thms: structured theory $\to$ set of sentences

thms$[\![$ prim-theory(Σ, S) $]\!]$ = the set of sentences provable from S
thms$[\![$ union(T, T') $]\!]$ = the set of sentences provable from thms$[\![$T$]\!]$ U thms$[\![$T'$]\!]$
thms$[\![$ rename(σ, T) $]\!]$ = the set of sentences provable from $\sigma^{\#}$(thms$[\![$T$]\!]$)
thms$[\![$ inv-image(dσ, T) $]\!]$ = dσ^{-1}(thms$[\![$T$]\!]$) = { t | d$\sigma^{\#}$(t) $\in$ thms$[\![$T$]\!]$ }

The constructor *prim-theory* produces an ordinary LCF (*prim*itive) theory. We use binary union of theories rather than n-ary union as in ordinary LCF for the sake of simplicity.

Suppose Σlist is sig(List), i.e. the signature consisting of the types *list of α* and *bool* and the constants *nil, cons, head, tail, null, true, false* and *not* (together with the remaining boolean operators), and Slist is the following set of sentences:

head(cons(x, l)) = x
tail(cons(x, l)) = l
null(cons(x, l)) = false
null(nil) = true

and σ: sig(List) $\to$ sig(Stack) is as defined above; then

rename(σ, union(prim-theory(Σlist, Slist), Boolean))

is the structured theory Stack.

The choice of structuring operators is not at all arbitrary. We were heavily influenced by our previous experience with the Clear specification language [BG 77, 80]. It happens that the semantics of Clear can be expressed entirely in terms of these simple theory-building operators (see [San 82a] for details). The theory-building operators of Clear are at a slightly higher level than those we have here; typically an application of a single Clear operator is equivalent to the application of two or three of our operators.

4 Parameterised theories

One feature which Clear has but which is missing here is a parameterisation mechanism. A *parameterised theory* (or procedure) in Clear can be viewed as a function taking a theory together with a signature morphism to a theory (parameterised theories with more than one argument are also allowed). Each parameterised theory has a formal parameter (itself a theory) which specifies the sort of actual parameter which the parameterised theory will accept. A typical example of a parameterised theory is Sorting, which produces a theory specifying a sorting function on lists of objects of type t, given a theory describing t. In this case the formal parameter would probably be the following theory:

```
POSet  =  the extension of Boolean by:
types      t
constants  ≤: t  ×  t  →  bool
axioms     x≤x  =  true
           x≤y and y≤x  ⟹  x = y
           x≤y and y≤z  ⟹  x≤z = true
```

This says that any actual parameter theory must include at least one type (other than bool) and a constant which satisfies the laws of a partial order relation on that type. Suppose we have an actual parameter, the theory SetNat of sets of natural numbers which includes the constant $\subseteq$: *setnat* × *setnat* → *bool*, defined in the usual way. Before applying Sorting to SetNat, we must construct a signature morphism which 'fits' the signature of POSet to that of SetNat. Suppose σ: sig(POSet) → sig(SetNat) maps the type *t* to *setnat*, and maps the constant $\leq$ to $\subseteq$ (and maps bool and its constants to themselves). Now the expression Sorting(SetNat[σ]) is legal and produces the desired result if the axioms of POSet (translated via $\sigma^{\#}$) are theorems of SetNat.

We have a (rather tentative and untested) scheme for introducing Clear-style parameterised theories into LCF. Let *apply* be the following function:

```
apply:  structured theory  ×  structured theory  →
                   structured theory  ×  signature morphism  →  structured theory

let apply (Proc,Formal) (Actual,σ) =
   if thms⟦rename(σ,Formal)⟧⊈thms⟦Actual⟧  then fail
   else let σ̂ = extend(σ,sig⟦Proc⟧) in
        union( union(Actual,rename(σ,Formal)),  rename(σ̂,Proc) )
```

This definition is rather high-level; in particular, implementing the first line requires a theorem prover. The auxiliary function *extend* takes a signature morphism $\sigma:\Sigma\rightarrow\Sigma'$ and a signature Σ'' (with $\Sigma\subseteq\Sigma''$) and returns a signature morphism $\hat{\sigma}:\Sigma''\rightarrow\Sigma'\cup(\Sigma''-\Sigma)$ which is the extension of σ to Σ'' by the identity (i.e. $\hat{\sigma}|_{\Sigma}=\sigma$ and $\hat{\sigma}|_{\Sigma''-\Sigma}=\mathrm{id}$). This assumes that LCF is modified to allow the same constant to have different types in different theories; otherwise $\hat{\sigma}$ could map each constant ω in Proc to the constant ω.*tag*, where *tag* is a token supplied by the user as an extra argument of *apply(Proc, Formal)*.

Apply is a general function for constructing parameterised theories having one argument (the generalisation to multiple arguments requires more mechanism). For example, let SortingTh be the following structured theory describing a sorting function on lists of objects of type t:

$$\begin{tikzcd}
 & & & \text{SortingTh} & \\
 & & {} \arrow[ur, hook] & & \text{sorting-extension} \arrow[ul, hook] \\
 & \text{POSet} \arrow[ur, hook] & & \text{List} \arrow[ul, hook] & \\
\text{poset-extension} \arrow[ur, hook] & & \text{Boolean} \arrow[ul, hook] \arrow[ur, hook] & & \text{list-extension} \arrow[ul, hook]
\end{tikzcd}$$

The *apply* function can be used to turn this *abstract* theory of sorting (it is abstract in the sense that nothing is known about objects of type t except that they are partially ordered) into an ML function:

Sorting: structured theory X signature morphism → structured theory
= apply (SortingTh, POSet)

If SetNat and σ:sig(POSet)→sig(SetNat) are defined as above, then evaluation of Sorting(SetNat, σ) produces the following result:

$$\begin{tikzcd}
 & & & \text{Sorting(SetNat}, \sigma) & & & \\
 & & & & & \text{SortingTh} \arrow[ull, hook, "\hat{\sigma}"'] & \\
 & & {} \arrow[uur, hook] & & {} \arrow[ur, hook] & & \text{sorting-extension} \arrow[ul, hook] \\
 & \text{SetNat} \arrow[ur, hook] & & \text{POSet} \arrow[ul, hook, "\sigma"] \arrow[ur, hook] & & \text{List} \arrow[ul, hook] & \\
\ldots \arrow[ur, hook] & \ldots \arrow[u, hook] & \text{poset-extension} \arrow[ur, hook] & & \text{Boolean} \arrow[ul, hook] \arrow[ur, hook] & & \text{list-extension} \arrow[ul, hook]
\end{tikzcd}$$

We would really like POSet to be an ancestor of SetNat in this result, since we have gone to the trouble of proving that the axioms of POSet hold in SetNat. We are exploring another view of structured theories (as 'decorated' diagrams in the category of theories) in which this would be more natural.

It is important to note several points regarding parameterised theories. First of all, adding parameterised theories does not add a new kind of structured theory constructor, since the result of applying a parameterised theory to an actual parameter is expressible using the present constructors. Second, this scheme for parameterising theories is only a suggestion inspired by Clear; other kinds of parameterisation may be useful as well. For example, MODLISP [DJ 80] permits ordinary values as parameters as well as theories. This is useful for defining (e.g.) the theory of n-dimensional vectors over a type t -- here, the theory defining t and the value n are both parameters. Finally, suppose A and B are both permissible actual parameters of Sorting (with fitting morphisms σ and σ' respectively). The structured theories Sorting(A, σ) and Sorting(B, σ') then share the parent SortingTh. This sharing will prove to be important later.

A different way of introducing parameterised theories into LCF was proposed by [LW 82], in which all the axioms of the formal parameter theory appear as assumptions of the axioms in the theory which results from the application, to be discharged in the normal fashion. This approach seems to be incompatible with our desire to retain the structure of theories; the result of an application could not have the parameterised theory or its formal parameter as ancestors.

5 Inference rules

As mentioned earlier, a structured theory inherits theorems from its ancestors in an indirect fashion. For example, to see if t is a theorem of *rename(σ,T)*, try to find a theorem t' of T such that $\sigma^{\#}(t') = t$ (this may involve proving a theorem in T). These relations between theories are reflected in the semantics of structured theories given above. In this section we give the LCF-style inference rules which encode the semantics and allow theorems in parent theories to be passed (often in an altered form) to their children.

In ordinary LCF we use the notation $A \vdash c$ to denote a theorem. We now need a different notation, since a theorem is not true in any absolute sense, but only relative to some theory. We will use the notation *(A⊢c) in T* to denote the assertion that $A \vdash c$ is a theorem of the structured theory T; note that *(A⊢c) in T* if and only if $A \vdash c \in \text{thms}[\![T]\!]$. We will call this a *fact*. The same trick is used to maintain the logical security of facts as ordinary LCF uses to protect theorems; *fact* is an abstract data type with the inference rules listed below as constructor functions.

PRIM-THEORY:	$s \in S$	$\Longrightarrow$	s **in** prim-theory(Σ, S)
UNIONLEFT:	s **in** T	$\Longrightarrow$	s **in** union(T, T')
UNIONRIGHT:	s **in** T'	$\Longrightarrow$	s **in** union(T, T')
RENAME:	s **in** T	$\Longrightarrow$	$\sigma^{\#}$(s) **in** rename(σ, T)
INV-IMAGE:	d$\sigma^{\#}$(s) **in** T	$\Longrightarrow$	s **in** inv-image(dσ, T)

In addition, the usual inference rules of PPLAMBDA (or whatever logical system we use) must be systematically modified to operate on facts rather than theorems. For example:

ASSUME: $w \vdash w$ **in** T
CONJ: $A_1 \vdash w_1$ **in** T and $A_2 \vdash w_2$ **in** T $\Longrightarrow$ $A_1 \cup A_2 \vdash w_1 \wedge w_2$ **in** T
SPEC: $A \vdash \forall x.w$ **in** T $\Longrightarrow$ $A \vdash w[t/x]$ **in** T   if t and x are of the same type

It is easy to prove from the semantics that these rules are sound. The following proof of the fact *(⊢ ∀x.isempty(pop(push(x,nilstack))) = true) in Stack* illustrates their use (we omit routine quantifier stripping):

	(⊢ null(nil) = true) **in** list-extension	(PRIM-THEORY)
$\Longrightarrow$	(⊢ null(nil) = true) **in** List	(UNIONRIGHT)
	(⊢ ∀l. ∀x. tail(cons(x, l)) = l) **in** list-extension	(PRIM-THEORY)
$\Longrightarrow$	(⊢ ∀l. ∀x. tail(cons(x, l)) = l) **in** List	(UNIONRIGHT)
$\Longrightarrow$	(⊢ ∀l. ∀x. null(tail(cons(x, l))) = null(l)) **in** List	(APTERM)
$\Longrightarrow$	(⊢ ∀x. null(tail(cons(x, nil))) = null(nil)) **in** List	(SPEC)
$\Longrightarrow$	(⊢ ∀x. null(tail(cons(x, nil))) = true) **in** List	(TRANS)
$\Longrightarrow$	(⊢ ∀x. isempty(pop(push(x, nilstack))) = true) **in** Stack	(RENAME)

Note that all of the real work of the proof is done by (the modified versions of) the usual PPLAMBDA inference rules. The new rules merely transport facts up the theory tree.

6 Tactics and strategies

The inference rules given in the last section could be used to prove facts in a 'forward' direction, but the preferred LCF style is to instead proceed backwards in a goal-directed fashion. A step consists of transforming the goal into a list of goals which, if they can be achieved (converted to facts), entail the desired fact. The transformation steps are carried out by backwards inference rules called *tactics*, which can be composed using *tacticals* to give *strategies*, as discussed earlier.

The following list contains tactics corresponding to each of the inference rules given in the last section. These are all simple ML programs, operating on goals of the form *(A|-c) in? T* and returning a list of goals (together with a proof, not shown).

```
PRIM-THEORYTAC:   s in? prim-theory(Σ,S)  ⟼  []  if s∈S, else fail
UNIONLEFTTAC:     s in? union(T,T')  ⟼  [ s in? T ]  if s is a sig⟦T⟧-sentence, else fail
UNIONRIGHTTAC:    s in? union(T,T')  ⟼  [ s in? T' ]  if s is a sig⟦T'⟧-sentence, else fail
RENAMETAC:        s  ⟼  s' in? rename(σ,T)  ⟼  [ s in? T ]  if σ#(s)=s', else fail
INV-IMAGETAC:     s in? inv-image(dσ,T)  ⟼  [ dσ#(s) in? T ]
```

Each of these tactics gives a way of *diving* into a structured theory with a sentence, yielding a goal concerning a parent theory and the (possibly transformed) sentence. UNIONRIGHTTAC and UNIONLEFTTAC choose different parents at a union theory; RENAMETAC yields a different result for the goal *s' in? rename(σ,T)* depending on which element of the set $\sigma^{-1}(s') = \{ s \mid \sigma^{\#}(s)=s' \}$ it is given. The following tacticals automate these choices:

```
UNIONTACTHEN:    tac  ⟼  (UNIONLEFTTAC THEN tac) ORELSE (UNIONRIGHTTAC THEN tac)
RENAMETACTHEN:   tac  ⟼  s' in? rename(σ,T)  ⟼
          ((RENAMETAC s_1 THEN tac) ORELSE ... ORELSE (RENAMETAC s_n THEN tac)) s' in? rename(σ,T)
             where {s_1 ... s_n} = σ^-1(s')
```

The standard LCF tactical *ORELSE*, given the two tactics tac_1 and tac_2, applies tac_1 to the goal unless it fails, in which case tac_2 is applied.

Each of the tactics above dives from a theory to one of its parent theories. The following composite tactical, given a tactic, explores the entire structured theory by diving repeatedly until it reaches a tip (a primitive theory). At this point the tactic provided as argument is applied. If this results in the empty goal list, then the goal is achieved; otherwise a failure is generated which is trapped at the most recent choice point (an application of UNIONTACTHEN or RENAMETACTHEN). The same process is then used to explore another branch of the tree (or the same branch, with a different sentence to prove), until the entire tree has been traversed.

```
DIVETAC:    tac  ⟼  g  ⟼
                if g = s in? prim-theory(Σ,S):  (TRY tac) g
                if g = s in? union(T,T'):  (UNIONTACTHEN DIVETAC tac) g
                if g = s in? rename(σ,T):  (RENAMETACTHEN DIVETAC tac) g
                if g = s in? inv-image(dσ,T):  (INV-IMAGETAC THEN DIVETAC tac) g
```

This uses an auxiliary tactical called TRY; it fails unless the tactic supplied is able to achieve the goal.

If *tac* is a powerful general-purpose proof strategy, then *DIVETAC tac* can automatically provide proofs for a wide range of facts. It dives down to the tip which contains the information needed to prove the fact at hand (finding the proper tip may involve a backtracking search), and uses *tac* to do the 'dirty work' of the proof.

This is quite a good way to go about proving facts in large structured theories. For example, if the goal is *(⊢p+q=q+p) in? T* where T is a structured theory describing a compiler, then almost all of the information buried in T is irrelevant and should be ignored lest the proof get bogged down by silly proof attempts. DIVETAC will fail quickly when attempting to follow most silly paths (going on to find the correct path) because of a mismatch between the sentence at hand and the signature of the irrelevant subtheory. For instance, consider the structured theory *union(Nat,Useless)*. An attempt to prove that p+q=q+p in the combined theory using DIVETAC will ignore the parent theory Useless; UNIONRIGHTTAC will fail immediately because *⊢p+q=q+p* is not a sig⟦Useless⟧-sentence. That is, provided that sig⟦Useless⟧ does not include the + operator. The *rename* construct can form a barrier to irrelevant goals in a similar fashion.

Unfortunately, a large class of facts remains which cannot be proved using DIVETAC. These are the cases in which there is not enough information in any *single* tip to accomplish the proof. For example, proving that the equation *length(append(l,k)) = length(l) + length(k)* holds in the theory of lists and natural numbers requires the use of information from both subtheories. DIVETAC will fail for this reason.

In cases like these, instead of diving into a structured theory with a sentence, we want to *dredge up* facts from the depths of the structured theory, forming the union of all the information available in all the ancestor theories. Then all these facts can be put to work in proving the sentence.

It is easy to prove the following derived inference rule:

DREDGE: $s \in \text{dredge}(T) \implies s \text{ in } T$

where dredge:

prim-theory(Σ, S)	$\longmapsto$ S
union(T,T')	$\longmapsto$ dredge(T) $\cup$ dredge(T')
rename(σ,T)	$\longmapsto$ $\sigma^{\#}$(dredge(T))
inv-image(dσ,T)	$\longmapsto$ dσ^{-1}(dredge(T))

Dredging does not retrieve *all* the facts available in a structured theory; some information may be lost along the way (in particular, it is hard to dredge in theories built using the *inv-image* constructor).

We add an extra component, the *list of available facts* to goals, with the notation *s* ***in?*** *T* ***using*** *F* to denote the goal *s* ***in?*** *T* with available facts *F*. DREDGETAC uses DREDGE to extract facts from the structured theory at hand, adding them to the list of available facts in the goal. Subsequent tactics can use these facts to help achieve the goal. For example, facts having the appropriate form can be added to the *simplification set* (another component of the goal) for use by the simplifier.

DREDGETAC: s' **in?** T **using** F $\longmapsto$ [s' **in?** T **using** [s_1 in T ... s_n in T] $\cup$ F]

where $\{s_1 \ldots s_n\}$ = dredge(T)

We have seen that DIVETAC is capable of proving a certain class of facts, yet DREDGETAC seems to be needed to collect the information necessary for the proofs of other facts. DREDGETAC alone is not capable of proving some of the facts which are handled with ease by DIVETAC, and besides it makes no use of theory structure. Some combination of diving and dredging seems to be necessary in a general strategy for proof in structured theories.

As mentioned above, often the structured theory at hand contains a great deal of information which is utterly irrelevant to the proof of a desired fact. It is important to restrict the available information as much as possible before attempting the proof using standard techniques. But how is our strategy to automatically determine exactly which subset of the available information is necessary for the proof of a fact? In the case of a ordinary LCF and conventional theorem provers where the axioms, previously proved theorems, etc. are stored in an unstructured form, the only approach seems to be some kind of heuristic filter which passes only 'relevant' facts. The construction of such a filter is difficult, for it is not always obvious which facts are relevant.

This problem is not so perplexing when we are given the information in a highly structured form, such as a structured theory. As observed earlier, it is easy when diving to exclude certain irrelevant subtheories entirely because *rename* and *union* constructs will form barriers to inappropriate goals. If the theory is well-structured, then it is likely that all of the information necessary to prove the fact will be located in a relatively small subtheory. DREDGETAC applied to this subtheory will normally collect all of the information necessary to prove the fact, without much that is irrelevant.

The following strategy is based on DIVETAC and DREDGETAC. The approach is to visit each node in the structured theory in precisely the same order as in DIVETAC, performing the same action at the tips. But after trying both parents of a *union* node and failing, DREDGETAC is used to attempt the proof in the combined theory. Hence dredging takes place on a theory only after all other methods have failed.

```
SUPERTAC:   tac ⟼ g ⟼
            if g = s' in? prim-theory(Σ,S):  (TRY tac) g
            if g = s in? union(T,T'):  ( (UNIONTACTHEN SUPERTAC tac)
                                        ORELSE (TRY (DREDGETAC THEN tac)) ) g
            if g = s in? rename(σ,T):  (RENAMETACTHEN SUPERTAC tac) g
            if g = s in? inv-image(σ,T):  (INV-IMAGETAC THEN SUPERTAC tac) g
```

There remains an important class of facts which cannot be proved using SUPERTAC. For example, in trying to prove *s* in the structured theory *union(T,inv-image(dσ,T'))* it might happen that *s* is neither a sig$[\![T]\!]$-sentence nor a sig$[\![$*inv-image(dσ,T')*$]\!]$-sentence, so diving is impossible. Furthermore, the proof of *s* might require the use of a fact *s' in inv-image(dσ,T')* which cannot be dredged -- perhaps $s' \in d\sigma^{-1}(s'')$, where *s"* follows from a and a' with $d\sigma^{-1}(a) = d\sigma^{-1}(a') = \emptyset$ but *s" in T'* is not explicitly available (it is not a previously proved fact). In cases like these it is necessary to first prove *s' in inv-image(dσ,T')* (or *s" in T'*) as a *lemma*. The idea for this lemma must come from the user or from some clever lemma-proposing tactic (but the problem of automatically proposing the right lemmas in such cases seems rather difficult).

Nelson and Oppen [NO 79] have described an elegant method for combining decision procedures for several independent theories into a decision procedure for the combined theory; this can be seen as an alternative to our DREDGETAC. Their method does not work when the theories share operators, so in general it cannot be applied to the union of structured theories. But in the special case where the theories do not share operators (and perhaps also for cases with certain restricted kinds of sharing) their algorithm could be applied in place of DREDGETAC.

The theorem prover of the ι (iota) system [NHN 80] also exploits the structure of specifications to facilitate proofs. It uses *theory-focusing* techniques [HN 79] which are related to the strategy embodied in SUPERTAC.

7 Implementation and future work

Most of the ideas in this paper were conceived during the construction of a system in LCF for proving theorems in Clear theories [San 82]. This system (written in ML) accepts a Clear theory expressed in terms of the theory-building operators described here (the conversion to this form is performed by a different program) and supports LCF-style theorem proving using inference rules, tactics and strategies similar to those discussed above. Recently this system has been modified to remove its Clear bias, and enhanced so that it contains the facilities presented here. Experimentation has so far been limited to a few relatively simple examples.

The parameterisation mechanism described above has not yet been implemented. Its implementation should present no problems, except that checking if a theory is a valid actual parameter must be implemented as a call on LCF itself to prove the necessary theorems.

The system does not currently remember the facts it proves for use as lemmas in later proofs. This would obviously be desirable, and should not be a difficult feature to implement. A related improvement would be to represent structured theories in such a way that common ancestors are truly shared, so that the addition of a newly-proved fact to an ancestor theory makes the fact available in the appropriate places throughout the entire structured theory. This is important (for instance) when we use parameterised theories. As mentioned earlier, if A and B are permissible actual parameters of the parameterised theory Sorting (for appropriate σ and σ') then Sorting(A, σ) and Sorting(B, σ') share the parent theory SortingTh (the analogous situation holds for any parameterised theory). It often happens that the proof of a fact in a theory such as Sorting(A, σ) will depend only on the information contained in SortingTh. (This in itself makes the proof easier, especially if A is large.) If the system remembers such a fact and sharing is implemented, then the fact will become available in Sorting(B, σ') as well. Such a sharing mechanism is already provided by LCF for conventional LCF theories.

One problem with the proposals presented in this paper is that the operations given for building structured theories are rather low-level. For example, in order to produce a structured theory which is the combination of T and T' enriched by some type operators S, constants Ω and axioms A (this corresponds to the only way of building new theories in conventional LCF) we must write:

```
union( union(T,T')
       prim-theory(<S,Ω> ∪ sig⟦union(T,T')⟧, A) )
```

This seems a rather cumbersome way of expressing a simple and commonly required operation.

Our first solution is to provide a function which makes enriching a theory easier. An infix function *enriched by* is defined which allows the example above to be written:

```
union(T,T') enriched by (S, Ω, A)
```

However, the structure which this hides is still visible during proofs. Ultimately we would prefer to use Clear's theory-building operations themselves as primitive theory constructors. Inference rules and tactics similar to those presented above can be developed for proving theorems in theories built in this way, although they will be somewhat more complicated than those given here. Our goal is to ultimately integrate Clear and LCF into a single system for specifying and proving theorems in large theories.

Acknowledgements

Thanks to Brian Monahan for help with LCF and for making helpful comments on a draft. Of course, we are indebted to Robin Milner and his colleagues for LCF and to Joe Goguen for his work on Clear. Thanks from DTS to Trattoria "dal Francese" in Norcia for an extra reason to attend CAAP 83. This work was supported by a studentship from the University of Edinburgh and by the Science and Engineering Research Council.

8 References

[ADJ 78] Thatcher, J.W., Wagner, E.G. and Wright, J.B. Data type specification: parameterization and the power of specification techniques. SIGACT 10th Annual Symp. on the Theory of Computing, San Diego, California.

[ASM 79] Abrial, J.R., Schuman, S.A. and Meyer, B. Specification language Z. Massachusetts Computer Associates Inc., Boston, Massachusetts.

[Bau 81] Bauer, F.L. *et al* (the CIP Language Group) Report on a wide spectrum language for program specification and development. Report TUM-I8104, Technische Univ. München.

[BG 77] Burstall, R.M. and Goguen, J.A. Putting theories together to make specifications. Proc. 5th Intl. Joint Conf. on Artificial Intelligence, Cambridge, Massachusetts, pp. 1045-1058.

[BG 80] Burstall, R.M. and Goguen, J.A. The semantics of Clear, a specification language. Proc. of Advanced Course on Abstract Software Specifications, Copenhagen. Springer Lecture Notes in Computer Science, Vol. 86, pp. 292-332.

[Cohn 79]	Cohn, A.J. Machine assisted proofs of recursion implementation. Ph.D. thesis, Dept. of Computer Science, Univ. of Edinburgh.
[DJ 80]	Davenport, J.H. and Jenks, R.D. MODLISP. Proc. 1980 LISP Conference, Stanford, California, pp. 65-74.
[GB 82]	Goguen, J.A. and Burstall, R.M. Institutions: logic and specification. Draft report, SRI International.
[GMW 79]	Gordon, M.J., Milner, A.J.R. and Wadsworth, C.P. Edinburgh LCF. Springer Lecture Notes in Computer Science, Vol. 78.
[GHM 78]	Guttag, J.V., Horowitz, E. and Musser, D.R. Abstract data types and software validation. CACM 21, 12 pp. 1048-1064.
[HN 79]	Honda, M. and Nakajima, R. Interactive theorem proving on hierarchically and modularly structured sets of very many axioms. Proc. 6th Intl. Joint Conf. on Artificial Intelligence, Tokyo, pp. 400-402.
[Les 82]	Leszczylowski, J. META SYSTEM. Preliminary draft report, Institute of Computer Science, Polish Academy of Sciences.
[LW 82]	Leszczylowski, J. and Wirsing, M. A system for reasoning within and about algebraic specifications. Proc. 5th Intl. Symp. on Programming, Turin. Springer Lecture Notes in Computer Science, Vol. 137, pp. 257-282.
[NHN 80]	Nakajima, R., Honda, M. and Nakahara, H. Hierarchical program specification and verification -- a many-sorted logical approach. Acta Informatica 14 pp. 135-155.
[NO 79]	Nelson, G. and Oppen, D.C. Simplification by cooperating decision procedures. TOPLAS 1, 2 pp. 245-257.
[San 82]	Sannella, D.T. Semantics, implementation and pragmatics of Clear, a program specification language. Ph.D. thesis, Dept. of Computer Science, Univ. of Edinburgh.
[San 82a]	Sannella, D.T. A new semantics for Clear. To appear in Acta Informatica. Also Report CSR-79-81, Dept. of Computer Science, Univ. of Edinburgh.