Virtual Theories*

Paul Curzon

University of Cambridge, Computer Laboratory
New Museums Site, Pembroke Street,
Cambridge. CB2 3QG. United Kingdom
Email: pc@cl.cam.ac.uk

Abstract. Proof is a programming activity. Consequently programming environments which support proof in the large are required. We describe an environment which supports one area of proof-in-the-large: that of theory management. We present the notion of virtual theories. They give the illusion of multiple active theories allowing the user to switch between different theories at will, proving theorems and making definitions in each. The system ensures that proofs only use resources that are available in the environment of the current virtual theory. The code has been implemented on top of the HOL90 system. A side effect is that a version of autoloading is obtained for HOL90. A more radical feature that is obtained is the autoloading of tools. The system has been tested on part of a real hardware verification proof.

Who controls the past controls the future,
Who controls the present controls the past.
George Orwell,
Nineteen Eighty-Four

1 Introduction

Interactive, machine-checked proof is essentially a programming activity. Proofs are programs in the meta-language of the theorem prover. Commands correspond to calls to proof tools. Thus many of the problems of managing large program developments apply to the development of proofs. Indeed the problems can be enhanced: with current state-of-the-art theorem proving technology, proofs are much longer than the system they verify. Most theorem proving systems provide some way of structuring proofs in the large. For example the HOL system provides a mechanism for grouping theorems about related definitions or constants into separate theories. This corresponds roughly to a module system of a programming language. As with programming, language features for structuring proofs are not the whole solution: programming environments must be provided to support the development process. Such a proof environment must supply tools to support many different activities. Typical systems give support for proof in

* In the Procceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications, LNCS, Springer-Verlag, 1995

the small. For example, they manage the proof obligations generated preventing them from being neglected and ensure that the proof programmer cannot corrupt the soundness of the system. They may also provide a means for working on several theorems at once: which is useful if a new lemma must be proved in the middle of some other proof. Environments that support proof in the large are also needed. In particular, similar tools for managing theories are needed. We describe a tool for managing one such aspect of theory development, that of switching between different theories in the middle of the proof effort. It is described in the context of the HOL system. This work was motivated by problems which arose on a real, large hardware verification project. The lack of proof support of this kind was severely hampering the proof effort.

2 Multiple Active Theories

When working on large proof efforts, it is often desirable to work on several theories at once, creating new theories and switching between old ones at will. For example, theorems about basic datatypes such as lists are often proved as lemmas to the main results of a theory. Ideally, such lemmas should be placed in a more appropriate theory. The current provisions of HOL (extend_theory, close_theory, etc) are at best clumsy to use. They essentially only allow theories to be developed in a linear fashion. For large proofs this is insufficient. As a consequence, either theorems and definitions are left in inappropriate theories, or a time consuming effort is required to tidy up the theory hierarchy after the proof is finished.

A common way to use HOL is to type the proof commands into an emacs file to maintain a record of the proof, then cut and paste these commands into a HOL session. One way to overcome the above problems would therefore be to record the proofs of lemmas directly into the source file of the theory they will ultimately reside in. The whole theory hierarchy can then be rebuilt in batch mode overnight so that it is in a consistent state for the next days session. The problem with this approach is that it is very difficult to keep track of the context (essentially the ML environment) in which each theorem was developed. This is necessary to ensure the appropriate context is recreated in the different source files. It is easy to use resources inadvertently that were believed only to be used by other theories. This leads to a great deal of time being wasted fixing all the problems that arise when the source files are replayed. If the theory hierarchy requires several hours to rebuild (which is not uncommon) this is not practical.

In this paper we present a virtual theory mechanism which was developed to overcome these problems. The aim was to provide simple automated help that would enhance our current way of working, without changing the core HOL system. In particular, we wanted to keep track of the dependencies of each separate theory so that the above problems would not be encountered. We wanted to provide a simple practical way of combining new proofs with old theories. The system was intended to ensure that no necessary context would be missed from the source files.

The system and corresponding methodology was intended to obey three main design criteria.

1. It should allow multiple theories to be developed in one session.
2. It should be possible to later rebuild each theory independently from its source file, with no additional user intervention.
3. Extending a theory in a new session should be no different from extending it in the original session in which it was created.

The last criteria is not met by the current HOL system because normally to extend a theory at least part of the ML state used originally must be rebuilt: loading tools, theorems, etc.

3 Virtual Theories

Virtual theories are just views of normal theories that can coexist and be extended within a single session. A virtual theory is created for each separate theory that is to be modified in a session (whether it previously exists or not). It is represented by a datastructure recording the resources available to the theory. Only a single theory is active at any time, but the user can view and switch between different ones at will. Resources are accessed through this datastructure using *censor* functions, rather than being stored in ML variables.

The use of the virtual theory system does not significantly alter the way HOL is used. As theorems and definitions are added, the user creates and switches between (virtual) theories as appropriate. The source commands that allow the theories to be recreated are stored in files specific to those theories.

The system does not modify the real theory files (ie, the permanent record of the theories) during the session. This must be done later by reloading the source files in batch mode using an appropriate makefile. This ensures that the source files are consistent before a real theory is created. Because of the general nature of the ML programming language (the meta-language of HOL) the consistency of the system cannot otherwise be guaranteed: theorems could be leaked between theories [2]. The virtual theory system forces the user to repeat commands if they must appear in more than one source file. It is assumed that the user is disciplined in ensuring that the source files do contain an accurate record of the session. This does not impose any new burden. Furthermore, user interfaces are now being developed that could maintain the source files automatically, so this is not a problem in the long term. To prevent the user from getting confused over which is the current virtual theory (and thus storing resources to the wrong place) most commands take the current virtual theory name as an argument, and fail if it is incorrect.

3.1 An Example Session

As an illustration, we give below a session to develop a theory "wombat" about constant `WOMBAT`. In the middle of the session an extension to an existing theory

"mylist" is made. The use of virtual theories ensures that a clear separation of the two theories is made.

On starting a session, we first create the initial virtual theory we will work in. A dummy real theory will also be created automatically. We cut and paste these commands from a source file for theory "wombat".

```
new_vtheory "wombat";
```

We wish the new theory "wombat" to depend on a previous real theory "mylist", so we make the latter theory a virtual parent. This makes the basic list operators and theorems in that theory available.

```
new_vparent "wombat" "mylist";
```

We then make a definition in virtual theory "wombat". It will ultimately be stored in the real theory, "wombat".

```
new_vdefinition("wombat","WOMBAT", (--‘WOMBAT l1 l2 = SUM l1 + SUM l2‘--));
```

Having made the definition we can prove theorems about it, accessing theorems from ancestor theories using censor `THM`.

```
vstore_thm("wombat","WOMBAT_CONS",
      (--‘WOMBAT (CONS x l1) l2 = x + SUM l1 + SUM l2‘--),
      REWRITE_TAC[THM "WOMBAT",THM "SUM"]);
```

For the main theorem we realize we need a new definition and lemma about lists, which we decide ought really be placed in theory "mylist". We therefore open a virtual theory for the existing real theory "mylist" to allow us to extend it.

```
extend_vtheory "mylist";
```

At this point the definition, new constant and theorem in theory "wombat" are no longer visible. They cannot inadvertently be used in definitions or proofs of theorems within the theory "mylist".

We make the new definition and prove the theorem. Commands are now cut and pasted to and from the source file of theory "mylist".

```
new_vdefinition("mylist","SAPP",
      (--‘SAPP l1 l2 = SUM (APPEND l1 l2)‘--));

vstore_thm("mylist","SAPP_SUM",
      (--‘SAPP l1 l2 = SUM l1 + SUM l2‘--),
      REWRITE_TAC[THM "SUM_APPEND", THM "SAPP"]);
```

We then switch back to our main virtual theory, switching source files once more.

```
extend_vtheory "wombat";
```

The resources of theory "wombat" are visible again. So are the theorems of theory "mylist" because it was made a virtual parent. Thus we can prove our main theorem.

```
vstore_thm("wombat","WOMBAT_SAPP",
      (--‘WOMBAT l1 l2 = SAPP l1 l2‘--),
      REWRITE_TAC[THM "SAPP_SUM", THM "WOMBAT"]);
```

Finally we close the theories, and end the session.

The source file of theory "wombat" will consist of the following commands which were cut and pasted into the HOL session.

```
new_vtheory "wombat";

new_vparent "wombat" "mylist";

new_vdefinition("wombat","WOMBAT",
(--‘WOMBAT l1 l2 = SUM l1 + SUM l2‘--));

vstore_thm"wombat"("WOMBAT_CONS",
      (--‘WOMBAT (CONS x l1) l2 = x + SUM l1 + SUM l2‘--),
      REWRITE_TAC[THM "WOMBAT",THM "SUM"]);

vstore_thm("wombat","WOMBAT_SAPP",
      (--‘SAPP l1 l2 = WOMBAT l1 l2‘--),
      REWRITE_TAC[THM "SAPP_SUM", THM "WOMBAT"]);

close_vtheory();
```

The following commands will have been added to the source file for "mylist", prior to its call to `close_vtheory`.

```
new_vdefinition("mylist","SAPP",
      (--‘SAPP l1 l2 = SUM (APPEND l1 l2)‘--));

vstore_thm("mylist","SAPP_SUM",
      (--‘SAPP l1 l2 = SUM l1 + SUM l2‘--),
      REWRITE_TAC[THM "SUM_APPEND", THM "SAPP"]);
```

After updating the makefile for these entries, we remake the theories from the source files. It is only at this point that the theory files for "wombat" and "mylist" are created and updated, respectively.

In practice the need for the addition of a theorem to an existing theory would often be discovered in mid proof, assuming a top down proof methodology is followed. This would not pose any problems. The original proof would just be suspended and then resumed once the other theory had been extended and the correct theory returned to.

3.2 Censors

All non-global resources are accessed using *censors*. Each kind of resource (theorem, tool, etc) has an associated censor which is used to access individual

resources of that kind. For example `THM`, used in the example session, censors theorems. Censors are functions which ensure that only resources that have been declared to the current virtual theory can be used.

Whenever a new resource is created, such as when a new theorem is proved, it must be declared to the censor. This is done using commands such as `vstore_thm`. The resource is recorded in the current virtual theory's datastructure. In the current implementation, the virtual theory datastructure stores the following resources of a theory,

- its parents (whether real or virtual),
- its theorems and definitions,
- its constants (term and type),
- its tools (tactics, inference rules, etc), and
- a cache of the resources accessed from other theories in this session.

For a complete system, other kinds of resources, such as libraries, need to be added. We have so far included only those needed for the particular case study we were concerned with.

When the censor is called to release a resource it attempts to recover it from the current virtual theory datastructure or that of its ancestors as determined from the datastructure. If the ancestors are real theories this may involve reading the theory file. If the resource cannot be found by the censor an exception is raised instead. If a resource is needed but not visible, either it must be redeclared, or the appropriate theory must be made an ancestor of the current theory. Thus if all declarations are recorded in the source file of the theory which is current at the time, the source file will be able to access all the necessary resources when it is replayed.

The virtual theory system requires a certain degree of discipline by the user. Resources must not be bound to top level ML identifiers, as otherwise the virtual theory manager will not be able to keep track of their use. Furthermore, the user must switch to an appropriate virtual theory before, for example, saving a theorem, or adding a new virtual parent so that a theorem can be accessed. If this is not done the same anarchy that results from working only with real theories can result.

4 Navigating Around Virtual Theories

New theories are created using `new_vtheory`. It adds a new empty virtual theory data object to a table of such objects maintained by the system. It also changes the current theory to be this new virtual theory. `new_vtheory` can be called as many times as necessary in a session as new theories are needed.

If an old theory, whether real or virtual, is required to be extended, a call to `extend_vtheory` is made. If the theory already exists as a virtual theory it is made current. Otherwise a new virtual theory is created. Real resources are only moved from the real theory (ie disk) into the virtual theory datastructure as they are needed since this is time-consuming. Read-only system theories can

new_vtheory s
Start a new virtual theory called `s`. If no working real theory has been created, then create one with the given name.

extend_vtheory s
Switch to virtual theory `s` if it exists. If it is not an existing virtual theory but is a real theory, open a virtual theory corresponding to it. If no working real theory has been created, then create one with the name "working".

new_vparent c s
Make the named theory (real or virtual) a virtual parent of `c` which should be the current virtual theory. If the named theory is not an ancestor of the current real theory, it will be made a real parent of that theory.

current_vtheory()
Return the name of the current virtual theory.

print_vtheory s
Give details of the named virtual theory.

close_vtheory()
Save the theorems proved in the current virtual theory to the working real theory.

Fig. 1. Navigating Around Virtual Theories

also be extended in this way. The real theory will not then be modifiable, of course, unless the source file can be written to by the user and the Makefile reflects the fact that the system will need to be rebuilt.

When the resources of some other theory that is not an ancestor of the current virtual theory are required, `new_vparent` is called. The ancestry of that parent is then made available to the censors when checking resources.

The user works in a dummy real theory which is created automatically by the first call to `new_vtheory` or `extend_vtheory`. Its purpose is to provide a working area within which the virtual theories live. All the definitions are saved to the dummy theory during the working session (since they must be saved somewhere). Similarly, all the real theories accessed in the session are made real ancestors of this theory. Its theory file is discarded after the working session, since everything of value is transferred to the source files of other theories. It is only later when the source files are replayed that the definitions and theorems are put into the real theory files.

5 Theorems

Theorems are saved, using `vsave_thm` or `vstore_thm`, in the virtual theory corresponding to the real theory in which they will ultimately reside. They are stored in a table mapping theorem names to stored theorems. The commands to prove them must be stored in its source file. Theorems are not bound to ML identifiers. They can instead be accessed using a call to the censor `LOAD_THM`, giving

LOAD_THM thy s
Return the theorem named `s` from virtual theory `thy`. Raise an exception if the theorem is not there or the theory is not a virtual ancestor. `LOAD_DEF` is similar.

THM s
Return the theorem or definition named `s`. It may be in the current virtual theory or in one of its virtual parents. In the latter case it will be autoloaded into the current virtual theory. `DEF` is similar.

vstore_thm(c,s,tm,tac)
Prove using tactic `tac` the goal given by `tm`. Store the resulting theorem in the current virtual theory with name `s`. Check `c` is the current virtual theory.

vsave_thm(c,s,th)
Store the theorem `th` in the current virtual theory with name `s`. Check `c` is the current virtual theory.

Fig. 2. Commands for manipulating theorems

the theorem's name and the ancestor virtual theory where it resides. `LOAD_THM` raises an exception if the theorem is not visible from the current virtual theory. Theorems should not be bound to ML identifiers. The only means used to access them should be via the censor. It will then only be possible to use those that are accessible in the current virtual theory.

5.1 Autoloading Theorems

HOL90 cannot do true HOL88 autoloading: ie, loading a theorem from an ancestor theory into the system when the the ML variable with its name is first used. This is because it would require the SML interpreter itself to recognise when an unknown SML variable had been referred to and take appropriate action to look for the theorem. In HOL90 the theorems must be manually loaded, though all the theorems of a theory can be loaded at once. This is time consuming and results in more theorems being loaded than are required. It also means a separate declaration must be made for each ancestor theory whose theorems are to be accessed in a session, even though from a logical perspective they are visible. Also, when a theory is extended in a new session, these declarations must be remade. The theory development cannot just be continued as though it had never been suspended. Autoloading is very useful for the large theory hierarchies encountered on large proofs, where remembering the names of theories where a theorem resides can be difficult, and theories can contain large numbers of theorems. Autoloading also helps with theorem mobility. If a theorem has been moved from one theory to another, then it can be automatically found. No change to other theories is needed provided it has been moved to one of their ancestor theories. Virtual theories do not give a complete solution to moving theorems, which would require a more complete tracking of dependency information, however.

new_vdefinition(c, s, tm)

Make the given definition `tm` in the current real theory. Also store it in the current virtual theory with name `s`. A check is made to ensure that only constants defined in the current virtual theory or its ancestors are used. `c` must be the current virtual theory.

Fig. 3. Making virtual definitions

With virtual theories, all theorems are obtained by a function call to a censor. Therefore if the named theorem has not previously been loaded, the function can automatically search for it throughout the theory's ancestry, and load it once found. This is implemented by a separate censor, `THM`. The autoloaded theorem can be loaded from both virtual and real theories. All that is required is that the theory is a (possibly virtual) ancestor of the current one. When the source files are rebuilt to create real theories, on first access, the theorem will be autoloaded from the real theory instead.

6 Constants and Definitions

Constants (term and type) are only visible to descendant theories of the theory they are created within. Thus with multiple active theories, censors must be used to ensure that constants are only visible to appropriate theories. We have implemented this by redefining the term and type parsers to check that the constants within the term or type are visible. This is sufficient provided the parsers are the only way used to create terms and types. Clearly, this could be circumvented using for example the term constructors directly, so in a complete system they should also be censored. However, censoring the parsers was sufficient for our immediate purposes.

Since the censors are likely to have to access many real theories when a term is first checked, the first use can be very slow. However, the names of all the constants of each theory visited are cached so that future accesses are quicker.

New constants are created using definitions. Thus the definition declaration functions must register the new constant they create as well as the corresponding theorem. Only simple term constant definitions have so far been implemented in the virtual theory system (see Figure 3) as this was all we immediately needed. However, the other forms of definition should not be problematic. The constants used in the definition must be checked by the constant censor, but as they are parsed by the term parser, this is done automatically.

Unlike the theorem censor, the constant censor does not need to return the resource (ie constant) on a successful check, thus the virtual theory datastructure records only the names of visible constants.

A minor advantage of not making ML bindings is that when giving definitions, the user no longer has to give the definition name twice as at present – once as the ML variable name and once as the name it will be stored as in the

new_vtool c s tool
Declare function `tool` to be available to the current virtual theory with name `s`. Check `c` is the current virtual theory.

LOAD_*TOOL* thy s
Return the function named `s` after checking that it is in the given theory and that that theory is an ancestor of the current virtual theory. Fail if it is not. *TOOL* is a name describing the type of the function (for example `THM_TACTIC` to find tools of type `:thm -> tactic`)

TOOL **s**
As `LOAD_`*TOOL* except that the tool is sought through the ancestors of the theory.

declare_tool_type s t
Create a censor for tools of type given by string `t`, using name `s`.

Fig. 4. Virtual Tools

theory. This removes the possibility of incompatibilities. It is also less confusing for learners of the system.

7 Tools

A theory's environment not only consists of its theorems and constants, but also the tools (tactics, inference rules etc) that it has access to. Thus tools need to be managed by the virtual theory system. Tools which exist in the core HOL system are not a problem since they can be accessed by all theories. Similarly, user defined tools that only refer to global resources can be made global and placed in a file that is loaded by all theories.

There are two issues which must be dealt with concerning other tools. Firstly, the code of the tool must be loaded before it can be used. Secondly, any resources it is dependent on (such as other tools and theorems) must be accessible. These issues must be accounted for in both the working session and the source files.

For tools dependent on theorems, this means that the appropriate theory must be an ancestor of the current theory. This is dealt with by ensuring that the tools only access theorems using censors. If the tool is used in a virtual theory where the theorem is not available, an exception will result.

Managing the loading of tools requires that the tools themselves be censored. If they are bound to ML variables when loaded, then all virtual theories in a session will be able to use them unchecked. They will not, consequently, necessarily be loaded when the source files are replayed.

As with theorems, when a tool is loaded it is declared to the censor (by a call to `new_vtool`) rather than being bound to an ML variable. Tools are accessed via function call, to a function `TOOL` say, rather than just by name. Unfortunately, the censor cannot store the tools themselves directly in a single datastructure. The problem is that it would not type check if say both a tactic and derived rule

were stored. One way round this would be to have a separate datastructure and related function for each tool type. This is obviously undesirable given the wide range of types that tools have and the flexibility HOL gives for tools with new types to be defined. Adding tools of types not envisaged by the virtual theory implementor would require reprogramming the virtual theory system.

The problem can be overcome using a method described by Larry Paulson [3]. It allows all tools of whatever type to be stored in a single datastructure. It uses the fact that exceptions can take arguments of different types, but the type of the exception does not reflect that argument's type. Its type is just `:exn`. A separate exception is declared for each tool type to be stored. Tools are then wrapped in their appropriate exceptions before being stored in the database. A separate function must be provided for each tool type to recover the tools (a single function will not type check). They take an exception argument which they raise and immediately handle, returning the embedded tool.

Thus to add a tool of a new type, the user must declare the exception, write a handler (the censor) and then declare the tool itself. This only involves adding functions, not changing existing code. Furthermore, the code required is of a very standard form. We have provided a function which creates the appropriate ML functions given strings representing the name of the censor to be created and the type it censors. The function does this by writing the code out into a file and then reloading it into the ML system. A concrete example is given below for theorem tactics.

To declare a new tool type for theorem tactics, `declare_tool_type` is called:

```
declare_tool_type "THM_TACTIC" "thm -> tactic";
```

This call loads in the ML code below, declaring an exception and two censors: one for loading from a given theory and one for autoloading (see Section 7.1). The functions `declare_new_tool_type` and `declare_to_load_new_tool_type` contain the code which does the censoring. Their argument is a raise-handle function for a specific exception.

```
exception DECLARE_THM_TACTIC of (thm -> tactic);

val THM_TACTIC = declare_new_tool_type
      (fn e => (raise e) handle (DECLARE_THM_TACTIC ttac) => ttac);

val LOAD_THM_TACTIC = declare_to_load_new_tool_type
      (fn e => (raise e) handle (DECLARE_THM_TACTIC ttac) => ttac);
```

New theorem tactics can now be declared to the censor.

```
new_vtool "wombat" "IMP_RES_REWRITE_TAC" (DECLARE_THM_TACTIC
      (fn th => IMP_RES_THEN (fn th1 => REWRITE_TAC[th1]) th));
```

The censor can then be used to recover and use the theorem tactic.

```
e(THM_TACTIC "IMP_RES_REWRITE_TAC" (THM "CONS"));
```

Censors for common tool types are pre-declared, so it is only unusual types that would require user intervention. The calls to recover tools are verbose, but this could be hidden by a user-interface.

The declaration of new tool types should also strictly be treated as a resource, since it involves ML commands which may need to be placed in more than one source file. However, for our purposes it was sufficient to treat them as global resources loaded in all sessions.

An advantage of using censors is that only a single copy of the code for a tool is needed. Frequently tools are developed that are both based on the theorems of a theory and used to prove theorems within that theory. This often means two copies of the tool's code are stored. One copy is placed in the source file for the theory. A second copy is placed in the file that is loaded when the tools are to be used in the development of other theories. Since censors are lazy, only one copy is now needed in the separate file. This file can be loaded before the theorems it uses have been proved. Only when a tool is actually used must its resources exist.

7.1 Autoloading Tools

Once accessed by censors, tools can be treated like theorems with respect to autoloading. Within the virtual world, any tool that has been declared in an ancestor virtual theory can be made accessible in the current theory. This poses a problem when migrating to real theories, however, as tools are not recorded in the theory files like the other resources. The source file of the descendent theory will have no record of the tools, and no way to access them from the real ancestor theory. A way is thus needed to associate tools with real theories as well as virtual ones. The most obvious place to make this association is in the theory files. However, that would involve changing core HOL. Instead, the virtual declarations of tools associated with a theory are placed in a file with the same name as the theory, but with a `.tsml` extension. Each such file also has an associated signature file (with `.tsig` extension). It just contains the names of the tools in the `.tsml` file and is used when searching for a tool. Only when the tool is found is the corresponding file of declarations loaded. A call to an unknown tool initiates a search through the ancestor theories of the current virtual theories to find it. Once found, all the code for that theory is loaded.

The granularity of loading is not very fine in this scheme. However, this could easily be overcome using multiple source files, with the signature file giving the file name for each tool. Tools could then be individually loaded. This has not been implemented yet, however.

If tools are to be used by theories other than their own, their declaration must not explicitly refer to a specific theory, but instead should refer to the current theory.

By associating tools with theories in this way, theories essentially become a form of lightweight mini-library. They have the advantage that tools are available, though not loaded, whenever the theorems and definitions they reason about are available. The location of the tools does not need to be known, they

are found by the system when used. If this system were adopted for the HOL system as a whole, the tools related to each datatype: booleans, lists, numbers, pairs etc, could be associated with the theorems for it, giving a more structured core system.

Autoloading is not to everyone's taste. Explicit loading is quicker, avoids name clash problems and leaves explicit dependency information in the source files. As with theorems, tools do not need to be autoloaded, however. Censors which take the name of the theory associated with the tool are provided. Furthermore, if the non-autoloading censor is used on the first call of a session, subsequent calls can then use the autoloaded form to quickly obtain the same tool because it will be cached in the virtual theory.

8 Use on the Fairisle Switching Fabric Proof

We were motivated to develop virtual theories because of difficulties encountered on a real proof of the Fairisle 16 by 16 switching fabric [1]. We had resorted to working in a single theory which we then tidied into separate theories at intervals. As a consequence the resulting theories were not tidied as well as they might, leaving a maintenance headache that was slowing the proof effort. The theory had grown very large and was desperately in need of tidying. The longer this had been put off, the worse the situation became. The result was that significant effort was required to tidy the proof.

The virtual theory mechanism was developed to prevent the same situation from recurring in subsequent proofs. It was also hoped that it would provide a way to tidy the existing theory. Initial results suggest that the latter aim has been achieved. We have tested the system by using it to tidy part of the theory. The script was about 6000 lines long and contained a wide range of theorems that should have been placed in many different theories. So far approximately 15% of the file has been tidied using the virtual theory system. The remainder is in progress. The effort involved was less than would have been required without virtual theories. The latter would have involved working with multiple HOL sessions and would probably have involved rebuilding parts of the theory hierarchy several times. From past experience of tidying scripts in that way, the virtual theory approach is easier, faster and more accurate (in the sense that only necessary dependencies are added).

9 Further Work

9.1 Other Resources

The current implementation of virtual theories is a prototype, and as such it only tracks the main resources such as theorems and tools. Various other resources would need to be dealt with to obtain a complete system. Essentially, anything that is part of the environment of a theory needs to be censored. Whenever a new resource is created it must be declared to a censor, and every time it is used

it must be accessed via that censor. The resources we considered were those relevant to our case study.

One important resource we did not explicitly consider is the libraries that a theory has access to. If a library is loaded for one virtual theory it will be available to (and so might unwittingly be used by) other virtual theories. It is not sufficient just to keep a list of libraries loaded by each theory, since some way is needed of detecting when the resources of the library are used. Also a way of declaring the resources when the library was loaded would be needed.

9.2 Interface Issues

We have used the virtual theory system in a cut and paste style, writing directly to a source file and copying the commands into the HOL session. This is typically the way HOL is currently used. It would be better if the source files were created automatically by the system. This can be done for normal theories using TkHolWorkbench [5]. It keeps a log of the session, yielding a file with the same contents that would be created using the cut-and-paste approach. It would be relatively simple to modify the interface for virtual theories and to direct the parts of sessions corresponding to different virtual theories to different log files.

We have experimented with a prototype interface for virtual theories based on TkHolWorkbench[2]. It graphically displays a tree of the virtual theory hierarchy, tracking any changes made. It also allows the user to switch to a new virtual theory within the HOL session by clicking on its icon in the tree. It could obviously be expanded to help visualise all the resources of a virtual theory. Such an interface could remove the problem of the verbosity of the virtual commands for retrieving resources. Once the interface knows of the source files it could provide means for hyper-text documentation of specifications, theorems and proofs to be displayed from the session.

The use of virtual theories would allow a dynamic theory browser such as TkHolWorkbench, to visualise all theories, rather than just the ancestors of the current theory. Other theories could be loaded into the session to be viewed without making them unwanted parents of an actual theory. This would be useful to allow theories to be searched prior to a decision to make them a parent.

For the purposes of exposition we have used distinct function names from those of real theories (for example, `new_vtheory` instead of `new_theory`). However, once working in the virtual world, there is no reason for the user to call the real versions (indeed this would be harmful). Thus the original functions could just be redefined when the virtual system is loaded.

9.3 ML Bindings

A key part of the virtual theory methodology as described is that resources such as theorems are not bound to ML identifiers and are only accessed via the censors. However, it may be possible to reintroduce variable bindings in a limited

[2] With lots of help from Donald Syme

form. Instead of an actual resource being bound, a censor specialised for that resource is bound. For theorems, this means a function taking a unit argument is bound (so they must be treated as tools). When the function is called the censor is checked before the resource is actually returned. Thus even though the variable bindings are visible to all virtual theories, their use in a theory where the resource is not visible will raise an exception

Autoloading is no longer possible as the ML binding must be made before it can be used. Also when a theory is extended, the ML bindings of the theory must first be set up, thus breaking our third design criteria. However, since this corresponds to the current situation in HOL90, a satisfactory methodology could perhaps be built around it, allowing multiple active theories, without the need to explicitly call censors.

9.4 Multiple Active Real Theories

In the long term it would be better if the sources did not need to be replayed to update the theory files. However, this would require a method of preventing leaks between theories due to ML bindings, intentional or otherwise. Such leaks are possible with virtual theories if the user does not follow the prescribed methodology of never making bindings of raw resources to ML objects.

One way real multiple active theories might be achieved is by using a meta-language which supports multiple processes so that each theory could have its own private meta-language environment. Alternatively, for theory development, rather than system development, a restricted meta-language interface could be provided which does not provide features that could lead to information leakage. This might be adequate in a commercial environment where the theorem prover was just being used to prove theorems rather than build new verification systems. A further possibility might be to build the censors into the primitives, tagging all resources with their environment. More research is needed in this area. In the meantime virtual theories provide a workable solution to the problem.

10 Related Work

Some of the functionality of virtual theories was implemented by Konrad Slind as part of his parametrised proof manager [4]. The main difference is one of focus. Slind was mainly concerned with tracking dependencies between theorems proved in a particular theory. We have been concerned with tracking all the resources used by a theory, and in particular the external resources, to allow a current proof session to be integrated with previous ones. Because in Slind's approach the proof manager keeps track of the dependencies, it is restricted to being used only with specially designed proof managers. In our approach, any proof manager, whether forwards or backwards can be used with no modification. The details of the implementation also differ. In particular, we keep track of dependencies by requiring that the resources are accessed by function calls, whereas Slind uses the proof manager to determine the resources used from the

commands passed to it. This requires more work on the part of the proof manager. It is clear that the functionality of our interface could be provided within the parametrised proof manager framework.

The abstract theory mechanism of Elsa Gunter [2] shares similar problems to those of virtual theories: that of making resources visible to only some theories. With abstract theories the resources are the axioms of the abstract theory. Because of the similarity, it may be possible to provide a unified system supporting both. However, we have done no work in this area.

11 Conclusions

We have presented a mechanism for managing multiple theories in HOL. It is loaded on top of the existing system so it does not involve incompatible changes to HOL itself. It can be used with any proof manager and proof style and enhances existing ways of working. The virtual theory mechanism was developed to overcome a particular problem encountered on a real hardware verification proof: that of tidying the script. It appears to have solved that problem very successfully. In future, by using the virtual theory mechanism, the problem should not arise in the first place.

Acknowledgements

I am grateful to the members of the Cambridge Automated Reasoning Group for their support. Donald Syme has been an invaluable source of encouragement. Mark Staples told me about the exception programming trick. Brian Graham proof read a draft of this paper. The work was funded by EPSRC grants GR/G23654 and GR/K10294.

References

1. Paul Curzon. Tracking design changes with formal machine-checked proof. *The Computer Journal*, 38(2), 1995.
2. Elsa L. Gunter. The implementation and use of abstract theories in HOL. In *Proceedings of the Third HOL Users Meeting*, 1990.
3. Larry Paulson. exn as dynamic type. Message sent to the comp.lang.ml newsgroup, January 1995.
4. Konrad Slind. A parameterized proof manager. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications: 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*, pages 407–423. Springer-Verlag, September 1994.
5. Donald Syme. A new interface for HOL - ideas, issues and implementation. In *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications*, Lecture Notes in Computer Science, 1995.