Stateless HOL

Freek Wiedijk
Institute for Computing and Information Sciences
Radboud University Nijmegen
Heyendaalseweg 135, 6525 AJ Nijmegen, The Netherlands
freek@cs.ru.nl

Dedicated to Roel de Vrijer, in the tradition of Automath.

We present a version of the HOL Light system that supports undoing definitions in such a way that this does not compromise the soundness of the logic. In our system the code that keeps track of the constants that have been defined thus far has been moved out of the kernel. This means that the kernel now is purely functional.

The changes to the system are small. All existing HOL Light developments can be run by the stateless system with only minor changes.

The basic principle behind the system is not to name constants by strings, but by *pairs* consisting of a string and a *definition*. This means that the data structures for the terms are all merged into one big graph. OCaml – the implementation language of the system – can use pointer equality to establish equality of data structures fast. This allows the system to run at acceptable speeds. Our system runs at about 85% of the speed of the stateful version of HOL Light.

1 Introduction

1.1 Problem

This paper describes a modification to the kernel of John Harrison's HOL Light [6, 8] proof assistant. Proof assistants are the best route to *complete* reliability, both in abstract mathematics as well as for verification of computer systems. Among the proof assistants HOL [3] is one of the more popular systems (next to Coq, Isabelle, PVS and ACL2), and among the HOL implementations HOL Light is one of the most interesting ones. HOL Light has both been used for extensive verification of floating point algorithms at Intel [7, 9], as well as for impressive formalizations in mathematics [5, 11]. Furthermore, a quite precise model of the HOL Light kernel code has been formally proved correct [10].

HOL is a direct descendant of the pioneering LCF [4] system from the seventies. In both LCF and HOL the user is not interacting with the proof assistant through a system specific language, but instead interacts directly with the interpreter of the ML language in which the system has been programmed. In the case of HOL Light this is the OCaml [13] language of Xavier Leroy.

For this reason in HOL there is no one keeping track of which theorems still are valid. Once a statement has been presented to the user as *proved* – by giving the user an element of the abstract datatype '`thm`' as a token of its being proven – it unavoidably will stay available to the user as a proved statement.

This approach has the advantage that the abstract datatypes of ML make it easy for the system to have a small proof checking *kernel* – also called *logical core* – with the property that if the code of that kernel can be trusted (and it implements a consistent logic), then it is certain that the system will be mathematically sound, i.e., it will not be possible to prove the statement of falsity $\bot$ in it. However, this approach has the disadvantage that it is difficult to *undo* state-modifying actions in the system.

T. Hirschowitz (Ed.): Types for Proofs and Programs 2009 (TYPES 2009).
EPTCS 53, 2011, pp. 47–61, doi:10.4204/EPTCS.53.4

In particular, in the existing HOL Light system it is not possible to change the definition of a defined constant:

```
# let X0 = new_definition ‘X = 0‘;;
val ( X0 ) : thm = |- X = 0
# let X1 = new_definition ‘X = 1‘;;
Exception: Failure "new_basic_definition".
```

In current practice, a user of HOL Light who wants to modify a definition will just reload the whole formalization. This is generally not a big problem, but can become quite slow. Also, reloading a formalization that is not finished yet and consists of bits and pieces that have been loaded manually, can be cumbersome.

There is a good reason that HOL does not have a way of undoing definitions. Let us suppose it would have a function `undo_definition`. Then we could have the following session:

```
# let X0 = new_definition ‘X = 0‘;;
val ( X0 ) : thm = |- X = 0
# undo_definition "X";;
val it : unit = ()
# let X1 = new_definition ‘X = 1‘;;
val ( X1 ) : thm = |- X = 1
# TRANS (SYM X0) X1;;
val it : thm = |- 0 = 1
```

Of course the ‘theorem’ `X0` will no longer be valid after we undo the definition of `X` as `0`, but there is no way for us to take away `X0` from the user once he or she has it. Hence, the system with `undo_definition` clearly is inconsistent, as one can prove `0 = 1` in it.

The problem is that the kernel of HOL Light keeps track of which constants have been defined already. It has *state*. Or, stated differently, it is not purely functional. To be able to undo definitions of constants in the HOL system, we will switch to a *stateless* kernel for the system.

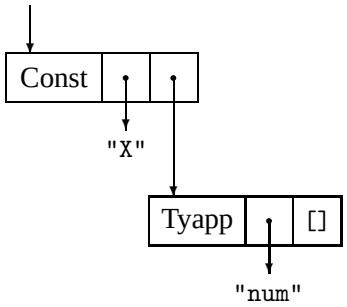
1.2 Approach

The approach that we will follow is quite simple. In HOL traditionally constants are *named*, i.e., they are identified by a string of OCaml type `string`. We will change this to identifying the constants by the definition itself.[1] That way, we do not need a state anymore to find the properties of the constant from the name. Then the properties will *be* the name.

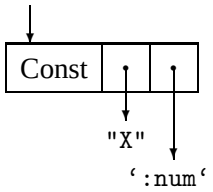
Actually, in our approach we will also include a string in the name of the constant, both for convenience and for backward compatibility. This means that the ‘names’ that we will use for constants will consist of a *pair* of a string and a definition. It is essential for efficiency reasons that when comparing constants the strings will be compared first. For this reason we put the string as the first component of the pair.

We will now look at what this means for the data structures in memory. Here is what the constant ‘`X`‘ from the above example looks like in memory in the traditional stateful HOL Light system:

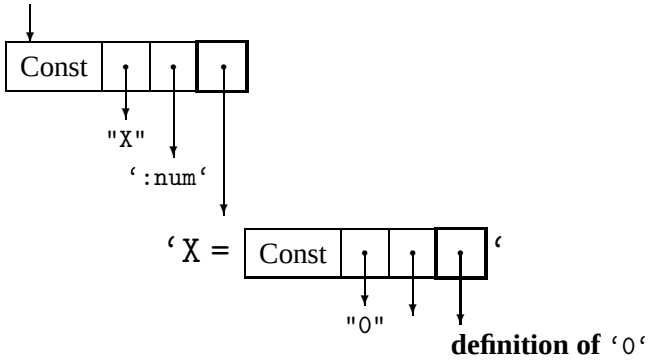
[1] In several systems the definition of a constant consists of two parts: first the constant is introduced and then the value of the constant is set. The approach from this paper does not allow such a separation.



This data structure takes 40 bytes in memory (24 bytes for the two blocks and 16 for the two strings), so this is not a large data structure. By hiding the detail of the `‘:num‘` type we can abbreviate this as:



Now here is the corresponding data structure for the constant `‘X‘` in our stateless HOL Light implementation:[2]



We added an extra field to the `Const` block that points to the definition of the `X` constant.[3] However, this addition is recursive: the constant `0` occurring in the definition of `X` *also* has this pointer which points to the definition of `0`. This continues all the way until we get to pure lambda terms that do not involve constants at all. Only then will this chain of references end.

Clearly, the constant `‘X‘` now is a large graph consisting of blocks connected by pointers. Of course for different constants these graphs will not be disjoint. Therefore, all constants together should be considered to be part of a single large data structure in memory.[4]

[2] We are simplifying reality slightly here. In HOL Light the number `0` actually is the term `‘NUMERAL _0‘`, so it is not a constant as drawn in the picture. The term `_0` however *is* a constant with definition `‘_0 = mk_num IND_0‘`. Tracing all further definitions from `mk_num` and `IND_0` is an interesting exercise that we will not pursue here.

[3] Note that the ‘`X`’ that occurs on the left hand side of the defining equation is a variable and *not* the defined constant. This equation was the argument to `new_basic_definition`.

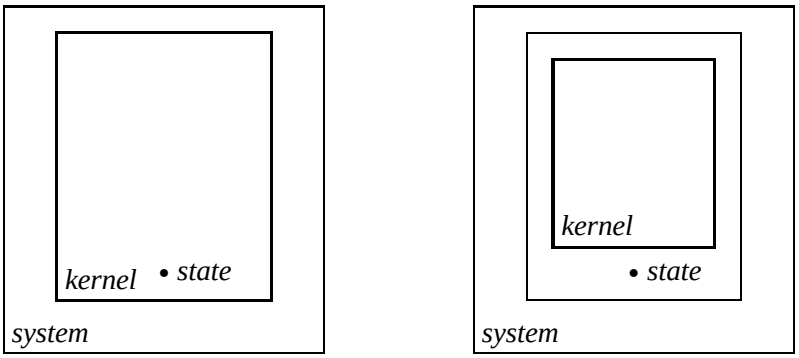
An alternative approach would be to have the new field just point to the right hand side of the equation. However, we considered it to be more elegant to have the field point to the exact data that was given to the function that introduced the constant to the system. In this case this is the argument to the function `new_basic_definition`.

[4] Note that although the amount of data for a single constant becomes much larger, the amount of data for the whole system stays roughly the same, as the data structures of the constants will be shared.

Despite the size of their data structures, comparison of constants still turns out to be rather cheap. In our modified system different constants actually still will have different strings in their names. Therefore, if constants are *not* equal, the comparison will fail exactly like it would fail before, i.e., while comparing the strings. However, if the constants *are* equal, their definitions in fact also will be equal, and even will be given by equal pointers. Now OCaml can be made to decide that two things are equal if the pointers to them are equal. Hence in the normal use of the HOL Light system constant comparison will never follow the pointers to the definitions.

However, if we use the `undo_definition` function from the above discussion, suddenly this does not hold anymore. The two `Xs`, that we defined in the example will then *not* be equal (despite the fact that their strings and types are equal) because the definitions will be different. This then will prevent the 'proof' of `0 = 1` from working.

Our stateless system is almost identical to the stateful one. We did not so much replace the kernel, as slightly change it. In particular we moved the stateful part outside the kernel. In a picture:



Stateful HOL Light *Stateless HOL Light*

Specifically, we split the implementation `fusion.ml` of the kernel (the thick box in the left picture) into a stateless part `core.ml` (the thick box in the right picture) and a stateful part `state.ml` (the part outside that box), as further described in Section 3. This means that our system still presents a fully compatible kernel to the rest of the system. Therefore all existing HOL Light developments will still work with the stateless kernel. The only code that needs to be adapted is code that probes the representation of constants. This hardly every happens, and even then the changes needed are small. We will discuss these changes in more detail below.

1.3 Related Work

The idea from this paper of having definitional information be part of the names of constants is applied to type theory in [2].

Many systems use an approach similar to the one that we describe here. For example the HOL4 system also uses a pointer internally to distinguish different versions of the 'same' constant. However the HOL4 system currently is not purely functional. The Matita and Epigram 1 systems use an approach similar to ours in that they *reconstruct* the context of a term from the term itself.

Many theorem provers have a purely functional kernel. This holds of course for all systems written in Haskell, but for example (since version 7) also for Coq [1]. However, one might argue that the kernels of these systems are not really without a notion of state, as in those systems *state* is an object that the kernel operates on. In Coq the state object is called an *environment*, while in HOL-based systems like ProofPower and Isabelle it is called a *theory*.

In Coq, one does not have a separate type for well-formed terms: to be purely functional it departs from the traditional LCF architecture in this respect. Only a type of *preterm* exists, that can be used to *extend* a state (at which time the preterm will be type checked). Specifically in the case of Coq the basic function of the kernel essentially is:[5]

```
add_constant : string -> preterm -> state -> state
```

This function adds a new constant with a given name and expansion to the state. Clearly the Coq kernel is functional by carrying the state around in a monadic fashion. In contrast, in our approach there does not exist a type corresponding to a state.

In ProofPower and Isabelle, *theorems* instead of constants are tagged, where the tag indicates which theory the theorem belongs to. Only theorems from compatible theories are accepted by the inference rules of those systems. ProofPower has stateful theories, and is quite similar to HOL4. Isabelle has a purely functional architecture, but for efficiency reasons implements it in a non-functional way [14]. It uses unique ids to provide an efficient approximation of the inclusion relation on theory content – both for efficiency and for decidability, since a theory may contain arbitrary data (including ML functions or fully abstract stuff).

The approaches of the various systems show a trade-off between easy access to the implementation of the system (systems like HOL) and ease of navigation of the formalizations (systems like Coq). We show that one does not need to sacrifice undo to get the accessibility of HOL's LCF architecture.

The trick of making the definitions part of the names of constants occurs in logic regularly. For instance, in Leisenring's book on the epsilon choice operator [12] the term $\varepsilon x.P[x]$ takes the place of a constant name for a witness of $\exists x.P[x]$. That way the completeness theorem can be proved without having to Skolemize first nor without having to add new constants.

1.4 Contribution

We present a version of the HOL Light system with the following properties:

- The kernel of the system is purely functional.
- The system supports undoing definitions in a logically sound way.
- The system is fully compatible with the existing HOL Light system: all existing developments can be run with minor changes.
- The system runs at almost the speed of the existing HOL Light system.
- The kernel of the system is theoretically easier to analyze.

1.5 Outline

The paper is structured as follows. In Section 2 we explain how we modified the HOL data structures and changed the kernel accordingly. In Section 3 we describe how the code of the system further had to be modified. In Section 4 we describe how to undo definitions. In Section 5 we discuss how to also have the kernel track in a functional way which axioms have been used for which theorems. In Section 6 we present the performance of our system. Finally, in Section 7 we conclude.

[5] In the actual Coq source code the `string` type is called `dir_path * label`, the `preterm` type is called `constr`, and the `state` type is called `safe_environment`.

2 The datatypes of the logical core

Here are the datatypes of the stateful HOL Light system (these are the traditional HOL datatypes):

```
type hol_type =
| Tyvar of string
| Tyapp of string * hol_type list

type term =
| Var of string * hol_type
| Const of string * hol_type
| Comb of term * term
| Abs of term * term

type thm =
| Sequent of term list * term
```

Of course, these datatypes are *abstract*, i.e., only the kernel can create data of these types. This is the essence of the LCF architecture. In our system, we changed these definitions to the following:

```
type hol_type =
| Tyvar of string
| Tyapp of hol_type_op * hol_type list

and term =
| Var of string * hol_type
| Const of string * hol_type * const_tag
| Comb of term * term
| Abs of term * term

and thm =
| Sequent of term list * term

and hol_type_op =
| Typrim of string * int
| Tydefined of string * int * thm

and const_tag =
| Prim
| Defined of term
| Mk_abstract of string * int * thm
| Dest_abstract of string * int * thm
```

That is, we *tagged* both the constants and the defined types by the information that was used to introduce them to the system. In the case of the types it turned out to be more convenient to integrate these tags with the string and arity into a `hol_type_op`.

The simplest example of this kind of tagging is the `Defined` tag for defined constants. However the constants for the functions that map between an 'abstract' defined type and the 'concrete' type from which it was carved have to be tagged too. (Those functions go in opposite directions: see [8] for a detailed description of the HOL type definition architecture.) For this we have the `Mk_abstract` and `Dest_abstract` tags.

In our implementation as a sanity check we temporarily replaced the underlined types by function types, to check that the system never followed the pointers corresponding to these fields (with this hack we did not have to change any other code in the system). That is, instead of `term` we used `unit -> term` and instead of `thm` we used `unit -> thm`. This works because OCaml will return `true` if a function is

compared to itself, while it will throw the `Invalid_argument "equal: functional value"` exception if it is compared to any other function.

There are various alternatives for the 'tagging data' in these type definitions. We chose to use the exact arguments of the functions that are used to define constants and types. However, more economical variants are possible. For example, instead of tagging the definition of a constant `X` with the defining equation `` `X = ...` ``, we could just have used the body of that definition. Similarly, in `Tydefined`, `Mk_abstract` and `Dest_abstract` we could have used a `term` instead of a `thm`. In that way we could have lost the dependency of `term` on `thm`. However, all these variants are really more or less equivalent, and we chose for the one where it is most obvious where the 'tagging information' originates.

One could also consider using structure-less nonces (`unit ref`s, say) as tags. However, this makes the system stateful again. Also, a formal analysis of the system will be harder that way, as in that case the tags will not be semantically meaningful.

We will now present how we adapted and reorganized the kernel according to these new datatype definitions. We only will do this for constants. Analogous changes had to made for defined types.

We will start by presenting the code before we changed it. The relevant functions for constants from the original, stateful HOL Light kernel are:

```
constants : unit -> (string * hol_type) list
definitions : unit -> thm list
get_const_type : string -> hol_type
new_constant : string * hol_type -> unit
new_basic_definition : term -> thm
mk_const : string * (hol_type * hol_type) list -> term
```

With `new_basic_definition` one introduces a new constant definition to the system, and with `mk_const` one creates constant terms. The implementation of these functions is completely straightforward:

```
let bool_ty = Tyapp("bool",[]);;

let aty = Tyvar "A";;

let the_term_constants =
   ref ["=",Tyapp("fun",[aty;Tyapp("fun",[aty;bool_ty])])];;

let constants() = !the_term_constants;;

let the_definitions = ref ([]:thm list);;

let definitions() = !the_definitions;;

let get_const_type s = assoc s (!the_term_constants);;

let new_constant(name,ty) =
  if can get_const_type name then
    failwith ("new_constant: constant "^name^" has already been declared")
  else the_term_constants := (name,ty)::(!the_term_constants);;

let new_basic_definition tm =
  match tm with
    Comb(Comb(Const("=",_),(Var(cname,ty) as l)),r) ->
      if not(freesin [] r) then failwith "new_definition: term not closed"
      else if not (subset (type_vars_in_term r) (tyvars ty))
      then failwith "new_definition: Type variables not reflected in constant"
      else let c = new_constant(cname,ty); Const(cname,ty) in
           let dth = Sequent([],safe_mk_eq c r) in
```

```
           the_definitions := dth::(!the_definitions); dth
  | _ -> failwith "new_basic_definition";;

let mk_const(name,theta) =
  let uty = try get_const_type name with Failure _ ->
    failwith "mk_const: not a constant name" in
  Const(name,type_subst theta uty);;
```

We will now show the corresponding code in our system. In our stateless version of HOL Light, the kernel interface becomes simpler. The only relevant functions now are:

```
new_prim_const : string * hol_type -> term
eq_term : hol_type -> term
new_defined_const : term -> term * thm
inst_const : term * (hol_type * hol_type) list -> term
```

with implementation:

```
let bool_tyop = Typrim("bool",0);;

let bool_ty = Tyapp(bool_tyop,[]);;

let new_prim_const(name,ty) =
  Const(name,ty,Prim);;

let eq_term ty =
  Const("=",Tyapp(Typrim("fun",2),[ty;Tyapp(Typrim("fun",2),[ty;bool_ty])]),
    Prim);;

let new_defined_const tm =
  match tm with
    Comb(Comb(Const("=",_,Prim),(Var(cname,ty) as l)),r) ->
      if not(freesin [] r) then failwith "new_definition: term not closed"
      else if not (subset (type_vars_in_term r) (tyvars ty))
      then failwith "new_definition: Type variables not reflected in constant"
      else let c = Const(cname,ty,Defined tm) in
           let dth = Sequent([],safe_mk_eq c r) in
           c,dth
  | _ -> failwith "new_basic_definition";;

let inst_const(tm,theta) =
  match tm with
  | Const(name,uty,tag) -> Const(name,type_subst theta uty,tag)
  | _ -> failwith "inst_const: not a constant";;
```

The remainder of the code was moved out of the kernel. These are the following functions:

```
the_term_constants : (string * term) list ref
the_definitions : thm list ref
get_const_type : string -> hol_type
new_constant' : string * term -> unit
new_constant : string * hol_type -> unit
new_basic_definition : term -> thm
mk_const : string * (hol_type * hol_type) list -> term
```

(We did not need to hide the stateful variables `the_term_constants` and `the_definitions` anymore, as changing them can no longer compromise the logic. For this reason we no longer need the `constants` and `definitions` functions for inspecting them.) The implementation of these functions again is straightforward:

```
let the_term_constants = ref ["=",eq_term aty];;

let the_definitions = ref ([]:thm list);;

let get_const_type s = type_of (assoc s (!the_term_constants));;

let new_constant'(name,c) =
  if can get_const_type name then
    failwith ("new_constant: constant "^name^" has already been declared")
  else the_term_constants := (name,c)::(!the_term_constants);;

let new_constant(name,ty) =
  new_constant'(name,new_prim_const(name,ty));;

let new_basic_definition tm =
  let c,dth = new_defined_const tm in
  match c with
  | Const(name,_,_) ->
      new_constant'(name,c); the_definitions := dth::(!the_definitions); dth;;

let mk_const(name,theta) =
  let tm = try assoc name (!the_term_constants) with Failure _ ->
    failwith "mk_const: not a constant name" in
  inst_const(tm,theta);;
```

Clearly, the code in our system is slightly more complicated, but essentially it is a reorganized version of the original code.

Our code has the property that makes it easy to distinguish calls to `new_constant` that add 'primitive' constants to the system – which in practice only is used for the Hilbert epsilon choice operator `‘(@)‘` – from other calls to `new_constant`. In our system the first kind corresponds to the kernel function `new_prim_const`.[6]

3 Modifications to the HOL source code

The sizes of the kernel files are compared in the following table:

	source file	*all lines*	*content*
Stateful HOL Light			
kernel	`fusion.ml`	669	394
Stateless HOL Light			
kernel	`core.ml`		383
state	`state.ml`		64
total			447

The last column counts the number of non-blank non-comment lines. (In our version of the code we removed all the comments, which means that the total line count of our files is not meaningful in comparison to the original system.) The kernel of the stateful HOL Light is called `fusion.ml` (it used to

[6] If one wants to be pedantic, one might keep track of calls to `new_prim_const` (and to its counterpart for types) in the `context` data structure described in Section 5 below. In some sense these are 'axiomatic' too.

be three files, `type.ml`, `term.ml` and `thm.ml`, which at some point were fused). We split this file in our kernel, called `core.ml`, and the remainder of the code, `state.ml`.

In the rest of the system only two kinds of changes had to be made. First, the system would be impractically slow if we compared objects using the default OCaml equality. Recently OCaml has been changed to consider Not-A-Number floating point numbers not to be equal to themselves, and for this reason the default equality test never uses pointer equality as an optimization. To get around this problem, we added the following lines at the start of the first HOL Light source file `lib.ml`:

```
let (=) = fun x y -> Pervasives.compare x y = 0;;
let (<>) = fun x y -> Pervasives.compare x y <> 0;;
let (<) = fun x y -> Pervasives.compare x y < 0;;
let (<=) = fun x y -> Pervasives.compare x y <= 0;;
let (>) = fun x y -> Pervasives.compare x y > 0;;
let (>=) = fun x y -> Pervasives.compare x y >= 0;;
```

Second, pattern matching on kernel datatypes had to be changed occasionally. As an example, in `basics.ml` the line

```
Tyapp("fun",[ty1;ty2]) -> (ty1,ty2)
```

had to be changed to

```
Tyapp(Typrim("fun",2),[ty1;ty2]) -> (ty1,ty2)
```

In the basic library of the system (which consists of 26,602 lines of source code) there were only 74 lines that had to be changed like this. These changes were systematic and could be made with a few global replacements.

4 Undoing definitions

With a stateless kernel implementing safe removal of definitions becomes trivial. We just add the following implementation of `undo_definition` to the source file `pair.ml` (right after the implementation of `new_definition`):

```
let undo_definition cname =
  the_term_constants := filter ((<>) cname o fst) !the_term_constants;
  the_core_definitions := filter ((<>) cname o fst o dest_const o fst o
      strip_comb o fst o dest_eq o snd o strip_forall o concl)
    !the_core_definitions;
  the_definitions := filter ((<>) cname o fst o dest_const o fst o
      strip_comb o fst o dest_eq o snd o strip_forall o concl)
    !the_definitions;;
```

This code has to be in (or after) `pair.ml`, because only there the variable `the_definitions` is introduced. In fact HOL Light has *two* variables with that name, one in the kernel (in our version of course in `state.ml` outside the kernel), and another one in `pair.ml`. We renamed the first one to `the_core_definitions`, and update both variables simultaneously.

Now with this function, our motivating example session from Section 1.1 runs as follows:

```
# let X0 = new_definition `X = 0`;;
val ( X0 ) : thm = |- X = 0
# undo_definition "X";;
val it : unit = ()
```

```
# let X1 = new_definition `X = 1`;;
val ( X1 ) : thm = |- X = 1
# TRANS (SYM X0) X1;;
Exception: Failure "TRANS".
```

As expected the system considers the two `X`s to be different, and does not allow the transitivity step anymore.

However, there still is a subtle issue. If we now print `X0` and `X1`, the system will do this in the following way:

```
# X0;;
val it : thm = |- X = 0
# X1;;
val it : thm = |- X = 1
```

I.e., the system prints out what appears to be contradictory judgements. Of course these judgements are not *actually* contradictory, the system is perfectly sound. The `X` in the first `thm` is the 'old' `X`, while the second is the 'new' `X`. It therefore will *not* be possible to prove from this that

```
val it : thm = |- 0 = 1
```

However one might ask, from a pragmatic point of view, how much difference that makes with the confusing printout of `X0` and `X1`.

This is not a problem with the consistency of the system, but with what in [15] is called *Pollack-consistency*. There is nothing wrong with the kernel of the system, but with its printing/parsing code. The statement of theorem `X0` is printed in a way that does *not* parse back to the same statement. That is (using the terminology from [15]) the printing/parsing functions are not *well-behaved*.

Of course, in [15] it is pointed out that the stateful version of HOL Light already was Pollack-inconsistent. Apparently this was not considered a serious problem, and the problem shown here might for the same reason be ignored. However (although we did not pursue this) in [15] a simple strategy is given to make a system Pollack-consistent, which can easily be applied here.

A simple variant of this would be to insert in the printing code some extra lines that check whether a constant that is being printed is equal to the 'current' value of that constant, and if not to throw an exception. In that way it is probably easy to have the system print `X0` as '`<obsolete theorem>`' (or something like that) after the definition of `X` has been undone. For this paper we were mainly interested in how to minimally modify the *kernel* of the system to find out what the performance of our approach would be (and not so much to further develop the result into a 'better' system), therefore we have not pursued implementing this yet.

5 Tracking the axioms

The stateful HOL Light system keeps track of the axioms that have been introduced by the user in the variable

```
the_axioms : thm list ref
```

We moved this variable out of the kernel, and therefore the system described thus far does not keep track of the axioms that have been used for the theorems. The whole system only uses three axioms, so one might not consider this to be a serious problem.

However, we also investigated a variant of the system where each `thm` was 'tagged' with the set of axioms from which it was derived. In that case each basic inference rule of the system had to take the union of this set of axioms for each of the `thm`s that it got as arguments. If implemented naively this would become expensive, computationally.

To get this to run with acceptable speed, we used the following data structure[7]. The `thm` type now is defined as

```
type context =
  int * term list array

type thm =
| Sequent of context * term list * term
```

The context type represents the axioms used in proving the `thm`. It consists of an array holding the *history* of the axiom lists during the execution of the system. Specifically it consists of an array of lists of axioms of decreasing length, prefixed with the length of the array minus one. The function that introduces an axiom now is:

```
let axiom_sequent ((n,axa) as ctx) tm =
  let axl = Array.get axa 0 in
  let ctx' = (n + 1),Array.of_list ((tm::axl)::Array.to_list axa) in
  let ax = Sequent(ctx',[],tm) in
  ax,ctx';;
```

Here the `(n,axa)` argument represents the set of axioms thus far. This is given by the stateful outside of the kernel. The code to merge contexts is:

```
let empty_context = 0,[|[]|];;

let merge_contexts ((n1,axa1) as ctx1) ((n2,axa2) as ctx2) =
  if ctx1 == ctx2 then ctx1 else
  if n1 < n2 then
    if Array.get axa1 0 = Array.get axa2 (n2 - n1) then ctx2 else
    failwith "merge_contexts" else
  if n1 > n2 then
    if Array.get axa1 (n1 - n2) = Array.get axa2 0 then ctx1 else
    failwith "merge_contexts" else
  failwith "merge_contexts";;
```

This code, when given two contexts, does not have to walk those contexts to see whether one is a prefix of the other (which would cost linear time), but instead uses the array data structure together with pointer equality to check whether the two contexts match (taking constant time). With this code only 'compatible' contexts, where one is a subset of the other, can be merged. Of course a more refined version of this code, that also is able to merge sets of axioms that are incompatible, could be written.

With this code, the 'set of axioms' for theorems always will be subsets of each other. We call this version of the system *with linear tracking of the axioms*. We were curious whether maybe there was a theorem that, for instance, only needed the first and third axioms but not the second one. For this reason, we made yet another modification to the code, that kept track of the *exact* set of axioms used. This version is called *with precise tracking of axioms*.

[7] One of the referees of this paper pointed out that the use of the `array` type introduces state to the kernel again, and that this undermines the point of the paper a bit. However, note that we use the arrays in a 'purely functional' way. We never update arrays, and only use them to be able to get to a specific index without having to walk a list.

In this version of the system we represented the set of axioms as a bit string in a 32 bit integer. (This version of the kernel therefore only can handle 32 axioms. As the actual system only uses 3 axioms, for the experiment this was sufficient.) Now the `context` type is:

```
type context =
  (int * term list array) * int32
```

and the `merge_context` code used OCaml's `Int32.logor` function on the `int32` bitstrings.

The result of this experiment however turned out to be that for *none* of the theorems in the basic library of HOL Light that are named by a global OCaml variable, a set of axioms is used that is not just a prefix of the list of the three axioms in the system. Therefore this refinement of the kernel turned out not to be very useful.

6 Performance

We measured the performance of our modified HOL Light versions, but only using wall clock time. Specifically we used the following code in an OCaml session:

```
#load "unix.cma";;
let starttime = ref (Unix.time());;
#use "hol.ml";;
Unix.time() -. !starttime;;
```

Here are the results on an unloaded Debian Etch system, using a computer with a single 1.86 GHz Intel Pentium M processor.

version	*running time*	*increase*
Stateful	113 s	
Stateless, without tracking of axioms	130 s	+ 15%
Stateless, with linear tracking of axioms	131 s	+ 16%
Stateless, with precise tracking of axioms	132 s	+ 17%

These are the times needed to load the basic HOL Light library. Of course, these numbers not only represent the time spend by the HOL Light system. For instance, displaying the output of the system in a terminal window already takes around 10 seconds. Still, the table gives a reasonable impression of the performance of the approach promoted in this paper.

7 Conclusion

7.1 Discussion

Switching HOL Light to our stateless kernel architecture has advantages and disadvantages: The advantages are:

- One gets the possibility to implement a function `undo_definition` in a logically sound way (and a similar function for type definitions).
- One gets a system that is probably easier to analyze theoretically. John Harrison's HOL in HOL formalization [10], in which he proves his kernel source code sound, currently leaves out the definitions. We expect that it will be easier to extend that work to include definitions for our version of the system, than it would be to do this for the stateful version of HOL Light.

It might seem that a system with 3 non-mutual datatypes is easier to analyze than a system with 5 mutually defined datatypes. However, it is much less difficult to reason about a purely functional program than about a stateful program. This more than compensates for the slightly more involved datatypes.

In the stateful version of HOL Light the semantics of data from the kernel depend on the state of the system, and often data has to be interpreted in a different state than in which it was created. This makes it hard to give the semantics in a compositional way. In contrast, in the stateless version all data has a direct interpretation, making analysis much simpler.

One subtlety with proving correctness of our stateless kernel is that it might be difficult to represent pointer equality in such a proof. Pointer equality is rarely considered in mechanized correctness proofs of functional programs. However, it is clear that pointer equality is just an optimization of structural equality and that proving the correctness of the kernel using structural equality could be considered equivalent.

The disadvantages of our system compared to the stateful HOL Light are:

- The system runs at about 85% of the normal speed in daily use.
- The kernel is more complicated. In particular the kernel datatype definitions are more involved.

We are undecided whether the slowness and added complexity in the kernel outweighs the nicety of having a purely functional kernel that supports undo.

7.2 Availability

A version of the system described in this paper can be downloaded on the web at the web address:

`http://www.cs.ru.nl/~freek/notes/hol_light-stateless.tar.gz`

This tar file contains just the basic library of the system, adapted for the stateless kernel. For reference, the tar file also contains the unmodified source code of the HOL Light version that we used for the experiment.

7.3 Future work

We mainly did this experiment to satisfy our curiosity, to find out whether the approach was viable. We were surprised that it all worked as painlessly as it did.

We did not argue here why our changes in the implementation are sound. Although this seems rather obvious, it would be good to have a formal analysis of this. An interesting way to do this would be to adapt John Harrison's HOL in HOL proof to also include definitions using our stateless variant of the code.

We would like the main version of the HOL Light system to adopt our stateless variant. In that sense, this article can be considered an open letter to John Harrison, asking him to consider doing this.

Acknowledgments. Thanks to Pierre Corbineau for interesting discussions on state in the HOL Light kernel. Thanks to Jean-Christophe Filliâtre, Rob Arthan and Makarius Wenzel for details of the architecture of the Coq, ProofPower and Isabelle kernels. Thanks to Randy Pollack for pointing out the Pollack-inconsistency issue addressed in Section 4. Thanks to anonymous referees for helpful comments.

References

[1] Jean-Christophe Filliâtre (2000): *Design of a proof assistant: Coq version 7*. Research Report 1369, LRI, Université Paris Sud.

[2] Herman Geuvers, Robbert Krebbers, James McKinna & Freek Wiedijk (2010): *Pure Type Systems without Explicit Contexts*. In Karl Crary & Marino Miculan, editors: *Proceedings 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice. Electronic Proceedings in Theoretical Computer Science* 34, pp. 53–67, doi:10.4204/EPTCS.34.6.

[3] Mike Gordon & Tom Melham, editors (1993): *Introduction to HOL*. Cambridge University Press, Cambridge.

[4] Mike Gordon, Robin Milner & Christopher Wadsworth (1979): *Edinburgh LCF: A Mechanised Logic of Computation*. *LNCS* 78, Springer Verlag, Berlin, Heidelberg, New York.

[5] Tom Hales (2007): *The Jordan Curve Theorem, Formally and Informally*. *The American Mathematical Monthly* 114, pp. 882–894.

[6] John Harrison (1996): *HOL Light: A Tutorial Introduction*. In Mandayam Srivas & Albert Camilleri, editors: *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96). LNCS* 1166, Springer-Verlag, pp. 265–269, doi:10.1007/BFb0031814.

[7] John Harrison (1999): *A Machine-Checked Theory of Floating Point Arithmetic*. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin & L. Thery, editors: *Theorem Proving in Higher Order Logics: TPHOLs '99. LNCS* 1690, pp. 113–130, doi:10.1007/3-540-48256-3_9.

[8] John Harrison (2000): *The HOL Light manual (1.1)*.

[9] John Harrison (2006): *Floating-Point Verification using Theorem Proving*. In: *Proceedings of SFM 2006, the 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems. LNCS* 2965, pp. 211–242, doi:10.1007/11757283_8.

[10] John Harrison (2006): *Towards self-verification of HOL Light*. In Ulrich Furbach & Natarajan Shankar, editors: *Proceedings of the Third International Joint Conference IJCAR 2006. LNCS* 4130, Springer, Seattle, WA, pp. 177–191, doi:10.1007/11814771_17.

[11] John Harrison (2008): *Formalizing an Analytic Proof of the Prime Number Theorem*. In R. Boulton, J. Hurd & K. Slind, editors: *Tools and Techniques for Verification of System Infrastructure*. The Royal Society, pp. 243–261, doi:10.1007/s10817-009-9145-6.

[12] A. C. Leisenring (1969): *Mathematical logic and Hilbert's ε-symbol*. Macdonald.

[13] Xavier Leroy et al. (2008): *The Objective Caml system release 3.11, Documentation and user's manual*.

[14] Makarius Wenzel (2002): *The Isabelle/Isar Reference Manual*. TU München.

[15] Freek Wiedijk (2010): *Pollack-inconsistency*. To be published in UITP 2010.