

Shallow Lazy Proofs

Hasan Amjad

University of Cambridge Computer Laboratory, William Gates Building, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK (e-mail: Hasan.Amjad@cl.cam.ac.uk)

Abstract. We show that delaying fully-expansive proof reconstruction for non-interactive decision procedures can result in a more efficient workflow. In contrast with earlier work, our approach to postponed proof does not require making deep changes to the theorem prover.

1 Introduction

Theorem proving programs serve to mechanise formal reasoning. Interactive theorem provers require user guidance to assist with formal proof. The proof proceeds by having the user type commands that tell the prover the next step in the proof. Such a step may invoke fully automatic decision procedures to handle the tedious but easier parts of the proof, leaving the user to focus on the parts requiring insight. Such decision procedures can often take a while to execute, taking up resources and making the user wait before the next command can be issued.

This situation is made worse for *fully-expansive* or *LCF-style* [10] theorem provers. This is because every proof must be constructed via the application of a small core set of primitive rules of inference. This has the great advantage that writing more powerful derived rules of inference cannot breach soundness, provided the core rules are sound. However, there is a speed penalty because of the large number of primitive inferences performed for each proof. Henceforth we consider only fully-expansive provers.

The standard way of making a decision procedure go faster is to execute it outside the theorem prover in an efficient manner (say as a C program), and then verify the result by proof in the theorem prover. Often the difference between the time taken to find a proof and to verify it justifies this approach. For instance, the result from a SAT-solver can be efficiently verified fully-expansively [7]. On the other hand, the result of a BDD operation is extremely inefficient to verify fully-expansively [11]. Most decision procedures fall somewhere along this spectrum.

For decision procedures that are hard to verify, the proof is usually done using a hybrid approach in which as much as possible of the proof search is done externally, resorting to internal proof if the trade-off between search and verification becomes unacceptable. In this way, proof search and verification proceed together in lock-step fashion.

In such a situation, it may help to postpone the justification underlying the verification, but to assert the verification right away so that the next search phase can proceed. This is supported by two observations:

- The verification can be done later when the user is otherwise occupied and computational resources are lying unused.
- Many of the lemmas for the verification proof may not eventually be required as the decision procedure may abandon a particular line of inquiry and thus all corresponding verification as well.

This approach was considered, most relevantly for us, by Boulton in his Ph.D. thesis [4], for the HOL88 theorem prover. The postponed proofs and theorems were termed *lazy*. In this paper we describe a slightly different approach to lazy proofs, which is less general but nonetheless useful in certain circumstances. We use the HOL-4 system [13, 9], a descendant of HOL88.

2 Background

To understand our approach, it is important to first know about the inference system of the HOL family of provers.

2.1 The HOL Inference System

We first give a quick overview of the inference system of HOL88 and HOL-4.¹ The logic of the prover is classical higher order logic, i.e. Church’s simple type theory [5], with Hindley-Milner polymorphism [16]. The inference system is based on natural deduction, though the choice of exactly which rules are primitive has been tempered by efficiency considerations. Hence some derivable rules are implemented as primitives. The term structure is simply-typed λ -calculus and the formula syntax is that of predicate logic.

The system is implemented in Moscow ML [19], a dialect of Standard ML [17]. The type of terms is `term` and the ML type constructors are available to users. A *goal* is an unproved theorem, consisting of a set of propositional terms Γ (the assumptions) and a propositional term A (the conclusion). Thus a goal has ML type `term set \times term`. A theorem $\Gamma \vdash A$ is a goal that can be derived via the primitive inference rules. Theorems have ML type `thm`. This is an abstract type, and so theorems can only be constructed by using the inference rules which are implemented as ML functions. Some of the primitive rules of inference we shall refer to are shown in Figure 1.

The complete set of primitive rules together with other functions that may affect soundness is often referred to as the *kernel* of the prover. The kernel is trusted code in the sense that a bug in the kernel can affect the soundness of derivations. Hence fully-expansive provers attempt to keep the kernel as small as is feasible.

¹ The differences are irrelevant to our discussion so we may regard the inference system as effectively being the same. For instance, HOL88 was actually implemented in Classic ML, but for the sake of clarity we’ll pretend it was written in Moscow ML.

Assumption:

$$\frac{}{\{A\} \vdash A} \text{ASSUME}$$

Modus ponens (A and A' are α -convertible):

$$\frac{\Gamma \vdash A \quad \Delta \vdash A' \Rightarrow B}{\Gamma \cup \Delta \vdash B} \text{MP}$$

Discharge (Δ is the (possibly empty) set of all terms in Γ α -convertible to B):

$$\frac{\Gamma \vdash A}{\Gamma - \Delta \vdash B \Rightarrow A} \text{DISCHB}$$

Generalisation (x is not free in the assumptions Γ):

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x.A} \text{GEN } x$$

Specialisation (y is free for x in A):

$$\frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[y/x]} \text{SPEC } y$$

Fig. 1. Some of the primitive inference rules of HOL-4

2.2 Deep Lazy Proofs

Since Boulton's method forms the starting point of our investigations, we provide an overview of it here. We shall call this method *deep* lazy proofs to distinguish it from our approach. The idea behind deep lazy proofs is to replace theorems with lazy theorems of abstract type `lazy_thm`.

The lazy theorems are lazy because they have not yet been proved, but may be used in proofs. Each consists of the underlying structure (i.e. the goal) of the theorem, together with a *justification function* of ML type `unit` \rightarrow `thm`. The justification function or *justifier*, is a closure which when executed returns a proved theorem of type `thm`. A lazy theorem is created by calling a special function `mk_lazy_thm` that takes a goal and a justifier and returns a lazy theorem.

This requires modifying all inference rules (including those in the kernel) to operate over `lazy_thm` rather than `thm`. The lazy inference rules would perform the required validity checks, internally modify the goal, and construct a new justifier that sequentially calls the justifiers of the argument lazy theorems before creating the required theorem.

This way any lazy theorem can be proved by executing its justifier. This execution will also result (because of the sequencing) in the execution of any justifiers for lazy theorems that were arguments to the inference rules that gave rise to the lazy theorem. A failed justification raises an exception. This ensures that all lazy theorems used in the lazy proof of the lazy theorem under consideration are indeed provable. The function `prove_lazy_thm` takes a lazy theorem,

executes the justifier, checks that the theorem it returned is indeed the goal, and returns it as a proved theorem of type `thm`.

Since the entire kernel is now lazy, all proof can be done using lazy theorems only. This results in a significant speed up since simply modifying the term structure of the goal and sequencing closures is in general much faster than calling an eager inference rule. This does not really apply at the level of primitive lazy inferences which effectively do as much work as their eager counterparts. The main benefit is for derived rules that can directly create lazy rules whose justifiers take a while to execute.

Proved theorems of type `thm` are still present in the system mainly because HOL88 was unable to persist closures, so lazy theorems had to be proved before theories could be saved between sessions. To allow these theorems to be reused in the main system, the type of lazy theorems was modified to optionally contain a proved theorem. A function was provided to convert a value of type `thm` to a value of type `lazy_thm`, the latter simply containing the proved theorem. This would enable saved theorems to be read back into the system in later sessions.

The deep approach has the obvious advantage of increased apparent speed (i.e. not including the justification time). Correctly used, it also results in increased real speed as lazy inferences applied in unused branches of proof search are not needlessly justified. Another big advantage is that the chosen representation does not impose any restrictions: deep lazy theorems can be used anywhere in the prover (except for storage).

The disadvantages are primarily from a developer’s perspective. First, this approach requires modifying the kernel, always a delicate enterprise in a fully-expansive prover. Second, unmodified derived rules do not become faster automatically. They have to be heavily modified to exploit lazy proof, and there is no guarantee that total execution time (i.e. lazy part plus justification) will be lower than the standard eager proof: it depends on the nature of the proof search the decision procedure performs.

Henceforth, we use the word “theorem” to mean both lazy and proved theorems, providing the appropriate qualification when necessary. We use the words “lazify” and “unlazify” to denote the operations of constructing a lazy theorem, and recovering a proved theorem from it.

3 A Shallow Approach

We felt the need for postponed proof during the development of a symbolic model checker embedded in HOL-4 [1]. The checker falls firmly in the “hard to verify” category, with BDD operations interleaved with proof steps. Since the model checker is completely non-interactive, being able to postpone expensive verification to, say, when the user is asleep,² or do it in the background, would

² Conversations with verification engineers at Intel Corp. and Microsoft Corp. indicate that model checkers are commonly run overnight, with office time spent setting up runs or analysing the output.

help significantly with the work-flow and some of the verification could possibly be discarded unused.

However, it is not clear whether deep lazy theorems are the answer. In particular, we do not wish to modify the kernel since we wish to retain compatibility with HOL-4. Hence the word “shallow” to contrast with Boulton’s method. This single restriction starts a chain of decisions that results in a rather different approach to lazy proofs.

For a start, we cannot have a separate type to represent lazy theorems. This is because the primitive rules operate over normal theorems, and thus so do all the derived rules. Having lazy derived rules that operate over a different type would be an interoperability nightmare. Whereas these derived lazy rules could conceivably use normal theorems via a suitable embedding of normal theorems into lazy ones (much like in the deep approach), the reverse is not possible unless the lazy theorem is first justified into a normal one. This would defeat the point of having lazy theorems, or else severely restrict their usefulness.

Also, since justifiers can now not be stored with the theorems (since we cannot modify the type `thm`), they have to be stored centrally, and a mapping ϕ between theorems and justifiers maintained. ϕ is a partial function, since a single theorem is not allowed two different justifiers, and proved (directly or unlazified) theorems have no justifiers. In ML, we implement ϕ by the look-up function on a splay map data structure.

3.1 Creating Lazy Theorems

Our solution³ is to have a lazy theorem for the goal (Γ, A) be represented by the theorem $\{A^g\} \vdash A$, where $A^g \equiv \forall x \in fv(A). A$ and fv computes the set of free variables of A . The mapping to justifiers is then given via terms by constructing ϕ so that $\phi(A^g)() = \Gamma \vdash A$. The only constraint is that

$$fv(A) \cap \bigcup_{\gamma \in \Gamma} fv(\gamma) = \emptyset$$

i.e., the conclusion and the assumptions must not have any free variables in common. The reason for this is discussed towards the end of §3.2.

$\{A^g\} \vdash A$ is derived by first constructing the term A^g and then proceeding

$$\frac{\overline{\{A^g\} \vdash A^g} \text{ ASSUME}}{\vdots \text{ SPEC}(x \in fv(A))} \{A^g\} \vdash A \tag{1}$$

where after the first inference we have a series of applications of the SPEC rule for each $x \in fv(A)$.

³ We note here that our first attempt based on the `mk_oracle_thm` function of HOL-4 failed because it was unable to cleanly track primitive inference rule applications.

Users are not allowed to modify ϕ directly. A function `make_lthm` is supplied that takes a goal, creates the required mapping in ϕ , and returns the lazy theorem. §3.3 demonstrates this on a simple example.

The reason we do not retain the assumptions Γ is practical: it does no harm, and retaining them complicates and slows down look-ups when evaluating ϕ . As noted in the next section, these assumptions are recovered from the justifier when the theorem is unlazified.

The reason for `make_lthm` creating $\{A^g\} \vdash A$ rather than $\{A\} \vdash A$ is to ensure that rules like `GEN` that would have succeeded on the proved theorem $\Gamma \vdash A$ do not fail on the lazy version. Indeed, some rule applications that would have failed on $\Gamma \vdash A$ succeed on $\{A^g\} \vdash A$. This would appear to suggest that we can prove stronger theorems with $\{A^g\} \vdash A$ than we could with $\Gamma \vdash A$. However, all such rule applications turn out to be vacuous (e.g. `GEN` $x t$ where x does not occur in t), so the “stronger” theorem is in fact derivable from $\Gamma \vdash A$.

3.2 Unlazifying Lazy Theorems

Now suppose we wish to prove B , and know that we require only $\Gamma \vdash A$ to prove it. We can use $\{A^g\} \vdash A$ instead of $\Gamma \vdash A$, since the former is a stronger theorem. The resulting “lazy” theorem is then $\{A^g\} \vdash B$.

In general, to unlazify a theorem $\Delta \vdash B$, we define the operation $(-)^s$ on terms to be such that $(A^g)^s = A$ and for each $C \in \Delta$ such that $\phi(C)$ is defined, do

$$\frac{\frac{\Delta \vdash B}{\Delta - C \vdash C \Rightarrow B} \text{ DISCH } C \quad \frac{\phi(C)()}{\Theta \vdash C^s} \text{ GEN } (x \in fv(C^s))}{(\Delta - C) \cup \Theta \vdash B} \text{ MP} \quad (2)$$

If ϕ is not defined, we move on to the next $C \in \Delta$ and repeat. We can access the contents of Δ since type destructors for `thm` are available.

Thus we eventually derive $\Delta' \cup \Theta' \vdash B$ as required, where Δ' contains all the *non-lazy* assumptions used in the proof of B , i.e. they were not introduced via `make_lthm`. Θ' similarly contains all non-lazy assumptions used in the proofs of the lazy theorems used in the proof of B .

As an example, consider the lazy theorem $\{A^g\} \vdash B$ described at the beginning of §3.2. Δ in this case is just $\{A^g\}$. Iterating over Δ , we consider $\{A^g\}$, and compute $\phi(A^g)()$ which gives $\Gamma \vdash A$, corresponding to $\Theta \vdash C^s$ in derivation 2 above. Now we generalise this by apply `GEN` for each $x \in fv(A)$, to get $\Gamma \vdash A^g$, corresponding to $\Theta \vdash C$ above. Now we take our lazy theorem $\{A^g\} \vdash B$ and use `DISCH` to get $\vdash \{A^g\} \Rightarrow B$, corresponding to $\Delta - C \vdash C \Rightarrow B$ above. Finally we use `MP` together with $\Gamma \vdash A^g$ to obtain $\Gamma \vdash B$, corresponding to $(\Delta - C) \cup \Theta \vdash B$. We give a more detailed and concrete example in §3.3.

A function `prove_lthm` is supplied that takes a theorem and discharges by the above strategy all assumptions in Δ that were added due to invocations

of `make_lthm`. The implementation of `prove_lthm` is a recursion rather than a simple iteration, since C^s may itself be the conclusion of a lazy theorem.

`prove_lthm` also modifies ϕ so that the next call $\phi(C)()$ will just return the result of the justification. This ensures that each unique lazy theorem's justification is carried out at most once. This is critical since often some fundamental lemmas are used hundreds of times in a single run of the decision procedure.

Note that the series of GEN rule applications will fail if any $x \in fv(C^s)$ occur free in Θ . This is the reason for having the constraint on free variables in §3.1. The assumptions added by `make_lthm` respect this constraint so lazy theorems may be used by the justifiers of other lazy theorems.

Note also that dropping the assumptions Γ in the goal of a lazy theorem during `make_lthm` does not matter, as any of them that do not get discharged (as they would if they were caused by a yet earlier `make_lthm` invocation), will appear as part of Θ' in the final theorem.

3.3 A Simple Example

We give a simple concrete example to illustrate our approach. Suppose we wish to prove that

$$\{\forall m.m + 0 = m\} \vdash (y = 0) \Rightarrow (x + y + z = x + z)$$

This can be proved directly using the decision procedures for linear arithmetic available in HOL-4. For the purposes of this example however, another way to prove this in HOL-4 is to use the simplifier together with the theorem

$$\{\forall m.m + 0 = m\} \vdash (y = 0) \Rightarrow (x + y = x)$$

Suppose the latter theorem can be proved by a HOL-4 proof procedure f that takes the goal as argument and returns the required theorem. Suppose further that calling f is expensive.

To postpone the expensive calls, we can create a lazy theorem. This is done by calling `make_lthm` with the goal $goal_1$ given by

$$(\{\forall m.m + 0 = m'\}, (y = 0) \Rightarrow (x + y = x)')$$

and the justifier $\lambda_. f(goal_1)$.⁴ `make_lthm` returns the lazy theorem

$$\{\forall xy.(y = 0) \Rightarrow (x + y = x)\} \vdash (y = 0) \Rightarrow (x + y = x)$$

and modifies ϕ so that $\phi(\forall xy.(y = 0) \Rightarrow (x + y = x)') = \lambda_. f(goal_1)$. Note that the assumption set $\{\forall m.m + 0 = m'\}$ has been dropped.

Now we prove our required theorem via simplifying with the lazy theorem, and get

$$\frac{\{\forall xy.(y = 0) \Rightarrow (x + y = x)\} \vdash (y = 0) \Rightarrow (x + y = x)}{\{\forall m.m + 0 = m\} \vdash (y = 0) \Rightarrow (x + y + z = x + z)}$$

⁴ We use quotes to distinguish terms of HOL-4's object logic from our metatheory. There is no such danger of confusion for theorems, since all theorems in this section are in the object logic.

Later, to unlazyify this theorem, we pass it to `prove_lthm`. `prove_lthm` iterates through the assumptions. In this case there is only one: $\forall xy.(y = 0) \Rightarrow (x + y = x)$. This is passed to ϕ which duly returns $\lambda_. f(goal_1)$. Executing this closure gives us $\{\forall m.m + 0 = m\} \vdash (y = 0) \Rightarrow (x + y = x)$. `prove_lthm` now recursively calls itself with this theorem, in case it is lazy. In this example it is not, so the recursive call returns the theorem itself. We now repeatedly use GEN to get

$$\{\forall m.m + 0 = m\} \vdash \forall xy.(y = 0) \Rightarrow (x + y = x)$$

Note that the dropped assumption has reappeared via the justifier.

Finally we use DISCH $\forall xy.(y = 0) \Rightarrow (x + y = x)$ on our target theorem to get

$$\vdash (\forall xy.(y = 0) \Rightarrow (x + y = x)) \Rightarrow (y = 0) \Rightarrow (x + y + z = x + z)$$

and then use the MP rule on this together with $\{\forall m.m + 0 = m\} \vdash \forall xy.(y = 0) \Rightarrow (x + y = x)$ from the previous paragraph to obtain

$$\{\forall m.m + 0 = m\} \vdash (y = 0) \Rightarrow (x + y + z = x + z)$$

as required.

3.4 Limitations and Benefits

This approach has a few problems not faced by the deep approach, which we discuss here:

1. Definitions cannot be lazified. This is because creating a definition introduces a fresh constant into the current theory, invalidating similarly named constants declared earlier. Thus the justifier's result will not match the corresponding assumption in the theorem to be unlazyified and failure will occur at the MP stage of derivation 2.
This is not such an issue because few decision procedures create definitions on-the-fly. If they do, the definitions are typically not recursive (used for abbreviating bigger terms say) and can be created very quickly.
2. The constraint in §3.1 means that our method cannot, in general, be used for interactive proof, except within a decision procedure that returns non-lazy results. However, often laziness in decision procedures allows some savings compared to eager execution, so its usefulness for interactive proofs is not entirely ruled out. Outside of interactive proof, this constraint has not been a problem in our experiments so far i.e. we have never needed to create a lazy theorem with assumptions that had free variables in common with the conclusion.
3. Any type variables in the conclusion of the lazy theorem appear in its assumptions as well. In theory, this is not a problem, since such type variables can be instantiated at the point of use. In practice, many derived rules in HOL attempt to instantiate type variables in the conclusion only and will fail with a lazy theorem which has those type variables in the assumptions (they must fail: unsoundness would result otherwise).

The only complete answer to this is to modify all derived rules to attempt full type instantiation whenever possible. This could result in loss of efficiency. Fortunately, most decision procedures tend to work mostly with ground theorems. Any theorems with type variables are usually general ones from pre-proved libraries. Such theorems are of course fully justified theorems and the problem does not exist.

4. ϕ is currently not constrained to be injective. This can create undesirable additional assumptions in an unlazified theorem, because the wrong justifier got called. Unlazifying would succeed since the justifier's result's conclusion is the desired one, but the final unlazified theorems would be stuck with the non-lazy assumptions of the justifier's result.

This is an engineering issue. Making ϕ injective will require the mapping translation to take assumptions into account, which could possibly be slow. In practice, this turns out to be a low priority issue since in all our test runs so far we have yet to come across a non-injective ϕ .

5. The series of GEN applications in derivation 2 can be very expensive if Θ contains very large terms.

This is a difficult problem. In practice, decision procedures often work with large terms, since the very reason for fully automated tools is to handle mountains of trivial proof. This problem can be solved by adding an iterated version of GEN to the kernel and deriving GEN as a non-primitive inference rule. We have verified this solution experimentally. However, modifying the kernel is not to be taken lightly. Fortunately, this does not become a big issue for our particular application (the embedded model checker) so we are content to live with it.

With the possible exception of the last, these are not really problems in practice, at least not for the problem domains we have considered so far. Unexpectedly, the fact that theorems tend to carry around a lot of assumptions while in lazy mode does not seem to have impacted performance. This may be because for our current problem domain, the number of dischargeable assumptions in the final lazy theorem rarely exceeds a hundred.

Within these constraints, we are able to create and use lazy theorems more or less as in the deep approach. We do have a few benefits as well:

1. The kernel is unmodified and so we are guaranteed soundness (modulo soundness of the kernel itself). This was a big plus during development when the ramifications of the method were still unclear.
2. No existing derived rule (or any other HOL-4 code for that matter) needs to be modified at all to work with this method. However, as with the deep approach, unmodified derived rules do not directly benefit.
3. There is only one type of theorem. Thus we do not have to implement translations between different types.

More importantly, we can unlazify a theorem at any stage. This provides some flexibility in implementing a more sophisticated system that would perhaps completely or partially unlazify theorems based on some metric that

measured the trade-offs for retaining the lazy version, at any stage during the decision procedure’s run. The closest analogy that comes to mind is that of a garbage collector in modern programming languages.

3.5 Refinements

We mention some refinements to the method which, for the sake of clarity, were not presented in §3.1 and §3.2.

First, as with the deep approach, we allow different modes of operation. Lazy mode works as described above. In eager mode, the justifiers are executed right away so there is no difference between eager mode and normal non-lazy execution. The mode can be changed at any time, and thus procedures can switch lazy proofs on or off depending on requirements.

Second, `make_lthm` is actually passed two closures rather than a goal and a single closure (i.e. the justifier). The first closure returns the goal, and the second is the justifier. This is so that eager mode does not pay the penalty for constructing the goal (which can sometimes be time consuming even in lazy mode), which is not required in eager mode. This does not guard against the goal being different from the justifier’s result. However, if such were the case, unlazyfying would fail at the MP stage of derivation 2, if not earlier.

Finally, the first closure passed to `make_lthm` does not just return a goal. It returns a goal and optionally a justifier which is used when unlazyfying. This is because it can use any information gained during the construction of the goal in lazy mode. Thus it could conceivably execute faster than the “eager” justifier (the second argument to `make_lthm`) that re-proves everything from scratch.

We have experimented with using shallow lazy proofs in our embedded model checker. The results are promising, as we show in the next two sections.

4 Lazifying a Decision Procedure

We chose to modify the HolCheck decision procedure [1] in HOL-4 to exploit lazy proof. The reason for this is our view that HolCheck could significantly benefit from such a development, as well as our familiarity with the code.

HolCheck is a symbolic model checker for the modal μ -calculus [15]. It is embedded in the HOL logic, with BDD operations considered atomic and executed externally for efficiency [8]. It is not possible to efficiently verify the part of proof search performed by the BDD operations. However, a model checker does more than BDD operations. It creates and applies transformations to models, translates between formalisms, and organizes the sequencing of BDD operations. All these operations are amenable to verification by formal proof.

Since model checkers are expected to be fast, it is in our interest to lower the overhead that is the verification part of the execution of HolCheck. For various reasons, it was found more efficient and more effective to carry out the verification interleaved with the proof search steps. The expectation is that by lazifying the verification, we retain the advantages and yet increase efficiency.

The model checking work-flow is somewhat different from the way interactive proof proceeds. The user sets up a run, and executes the checker. Most of the user’s time is then spent analysing the output of the checker to fix bugs in the model or the specification. Then the process is repeated. It is thus imperative that the checker run as fast as possible.

Our main hope was that by using lazy proofs, we could delay the relatively slower verification parts of the proof to some later time, say when the user is analysing the checker’s output, or overnight. This would greatly speed up the model checking work-flow, at no extra cost.

Currently we have lazified only the core of the model checker. This includes formal model construction, translation by proof from the specification logic CTL [3] to the model checker’s μ -calculus and back, the model checking engine itself and trace generation. The abstraction mechanisms have not yet been made lazy, and were turned off for benchmarking purposes.

The benchmarks are for model checking several properties for a 3x3 game of tic-tac-toe, and for a small pipelined ALU. The examples were chosen to be big enough to not finish execution in a couple of seconds, but not so big that the majority of the model checker’s time was spent in BDD operations. This was so that we could time the savings (if any) in the non-BDD verification part of the checker’s time.

Table 1. HolCheck benchmarks with and without lazy proof (as % of eager time)

	tic-tac-toe	ALU
Eager total (gc)	100 (30.13)	100 (34.28)
Lazy lazy	38.88	30.47
Lazy justification (overhead)	19.53 (7.39)	75.95 (4.98)
Lazy total (gc)	58.41 (6.21)	106.43 (4.76)

The results are given in Table 1. All numbers are normalised with respect to the topmost row, which represents the time taken by running HolCheck in eager or non-lazy mode, scaled to 100. The absolute numbers are not important. The benchmarks took from about 30 seconds to about a 1000 seconds to complete.

In the table, “gc” stands for the time taken by the Moscow ML garbage collector, and is included in the total time. “lazy” gives the time taken to run HolCheck in lazy mode, and “justification” indicates how long it takes to later unlazify all the lazy theorems produced. “overhead” shows the portion of time spent doing the work of derivations 1 and 2.

The HolCheck core’s verification part is already heavily optimized. Nonetheless, we see that the lazified system runs around thrice as fast as the eager one. Garbage collection overheads are also comparatively low, mainly because proof is delayed so that the number of terms that appear and disappear is low.

Factoring in the justification stage, the tic-tac-toe benchmark still runs almost twice as fast as the eager version, but the ALU benchmark is slightly slower. This is because in the tic-tac-toe benchmark, some of the properties

checked are designed to fail (to exercise the trace generation code). Hence unlazifying for those properties is not done, whereas the eager version expends extra time proving the corresponding verification conditions. In the ALU benchmark, all properties succeed, so all theorems must be unlazified. In this case, the small lag is explained by the (un)lazifying overhead, as expected.

Note that we omit the number of primitive inferences made in the different modes. This is mainly because the number of primitive inferences only very roughly correlates with time savings and thus primitive inference figures may overrate the benefits of lazy proofs. For instance, the lazy version of HolCheck performs about 80% fewer primitive inferences than the eager one (though the justification phase makes up for the difference), but is not five times as fast. Unlike the deep approach, our lazy mode is not primitive inference free: we do not bother lazifying proofs written directly in terms of primitive inferences since these do not save us any time. There were quite a few of these in the pre-lazified optimized HolCheck core.

The speed gains are remarkably similar to those achieved using the deep approach, which also reported roughly a three-fold increase in the speed of the lazy decision procedure, with justification not always saving any time. It remains to be investigated if this is more than coincidence. It should be kept in mind that the deep approach's results were reported over a decade ago, and theorem proving technology has improved much since then. Nonetheless, we have achieved similar performance gains without modifying the kernel of the prover, with room for improvement if we do.

5 Lazifying a Derived Rule

The primary beneficiaries of our approach are expected to be decision procedures. However, most decision procedures make use of derived rules of inference. It would thus be nice if we could over time build a library of lazified derived rules. This would ease the further development of lazy derived rules and decision procedures.

The expectation here is that the lazy derived rule will run faster than the eager one and even if the justification takes as long as the eager rule, it would not matter since the justification will be carried out at a time when the user is busy elsewhere.

In an experiment, we chose to lazify the GEN_PALPHA derived rule in HOL-4. This rule performs alpha conversion with respect to the binder of any abstraction, with the further ability to handle not only single variables, but paired structures as well. So,

$$\frac{\Gamma \vdash @x.t}{\Gamma \vdash @y.t[y/x]} \text{ GEN_PALPHA}y$$

where @ is any binder, and x and y are possibly paired structures of the same type. Variable renaming is performed if necessary to avoid variable capture within t .

Table 2. GEN_PALPHA lazy execution time (as % of eager time)

Number of variables\Term size	800-3000	1500-5500	2500-8000	3000-12000
100	55.81	55.1	57.14	53.33
200	22.46	22.52	24.62	23.72
300	13.1	13.45	14.96	17.03
400	9.66	9.77	12.28	12.49
500	5.82	7.15	9.59	9.88

Table 3. GEN_PALPHA justification time (as % of eager time)

Number of variables\Term size	800-3000	1500-5500	2500-8000	3000-12000
100	188.37	289.8	422.98	572.08
200	150.97	203.95	266.36	314.71
300	135.1	172.79	202.24	227.16
400	125.69	152.15	171.57	190.51
500	128.49	144.26	154.56	165.87

This rule was chosen for two reasons. First, it is used often in HolCheck and can be slow on large terms (not through any fault of the implementation). Second, it is not so low-level that lazifying has no benefit, and not so complicated that lazifying would not work because we do not understand how the rule works.

Lazifying this rule is not as simple as passing a goal and the eager version of GEN_PALPHA to `make_lthm`. We need to work out what the $t[y/x]$ part of the goal will look like. Simple substitution will not do since x and y may have the same type but different structure e.g. x could be the pair (a, b) with a and b occurring separately in t , while y is just a single variable. Also, some components of y may need renaming to avoid capture by internal binders in t . Fortunately, the code for this can be extracted without too much hardship from the code for GEN_PALPHA, though it is not as simple as a copy-paste operation.

Constructing the goal takes time, and this is where the motivation for the refinements mentioned in §3.5 comes from. Once that is done, we proceed with `make_lthm` as usual, using as our justifier the eager version of GEN_PALPHA. Later we intend to write our own justifier that can take advantage of any information gleaned during goal construction.

Performance evaluation results are given for lazy execution and justification in Table 2 and Table 3, with all times normalised to the eager version’s performance, which is set at 100 as before.

Both the eager and lazy versions of GEN_PALPHA were run on a number of randomly generated terms of various sizes, with varying numbers of free variables and levels of internal binding (to exercise the renaming aspects of the code). Term size is, roughly speaking, the number of abstractions and applications in the underlying λ -structure of a term. Term sizes are given as ranges since the randomly generated terms contained predicates on a random number of variables, hence the term size increased with the number of free variables.

Again, the absolute numbers are not important. Just to give the reader an idea, the eager version averaged about half a second with a 100 variables and a term size range of 800-3000, and about 120 seconds when the number of variables is increased to 500. Increasing term size only causes a linear degradation in eager performance however.

As expected, the lazy version runs much faster than the eager version, much more so than in the deep approach. However, the justification phase now takes much longer. The performance of the eager version scales well with term size, but degrades badly with increasing the number of variables, which is where the lazy version wins out. Justification degrades badly in both dimensions, though with higher variable sizes, the relative performance improves on account of the degradation in eager performance.

Since the justifier is just the eager version, all the extra time in justification is being taken up by the lazifying overhead. Profiling shows that 99% of this overhead can be traced to the series of GEN applications in derivation 2. This problem has already been noted. If an iterated version of GEN is added to the kernel, this overhead almost disappears.

Thus, at least in this case, using the lazy version is only useful if the justification is to be postponed until much later, rather than done immediately after the lazy rule finishes. This renders this lazy rule unfit for interactive use (without modifying the kernel) where, as noted in §3.4, only fully justified theorems are usable. However, it can still be used in decision procedures and indeed has been used in HolCheck.

6 Related Work and Conclusion

The work by Boulton [4] was the starting point of our research and his observations strongly influenced our decisions. Laziness has been exploited in formal verification elsewhere, for instance in incremental SAT-based abstraction for model checking [2, 6]. More relevantly, the lazy approach has been applied to the Nelson-Oppen congruence closure algorithm [12, 18].

Shallow lazy proofs succeed very well at justifying the observations made towards the end of §1, at least with respect to HolCheck and GEN_PALPHA. However, our experience with lazifying indicates that the gains are strongly dependent on the nature of the proof procedure under consideration.

In particular, the more proof is done externally, the less lazy proofs are likely to help, even in the lazy stage. Also, if the proof search is very efficient with few abandoned lines of search, the justification phase will make up for any time savings made during the lazy stage.

The constraints listed in §3.4 do restrict the applicability of our approach. However, in actual practice, the performance issues have not seriously affected efficiency, and the theoretical restrictions have not constrained shallow lazy proofs enough to affect usefulness.

This method is applicable to all theorem provers that support the rules of inference listed in Figure 1.

References

1. H. Amjad. Programming a symbolic model checker in a fully expansive theorem prover. In David A. Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2003.
2. Thomas Ball, Byron Cook, Shuvendu K. Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference*, volume 3114 of *LNCS*, pages 457–461. Springer, 2004.
3. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
4. Richard John Boulton. Efficiency in a fully-expansive theorem prover. Technical report, University of Cambridge Computer Laboratory, 1994.
5. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.
6. Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer-Verlag, July 2002.
7. M. J. C. Gordon. HolSatLib documentation. <http://www.cl.cam.ac.uk/users/mjcg/HolSatLib/HolSatLib.ps>, 2002.
8. M. J. C. Gordon. Programming combinations of deduction and BDD-based symbolic calculation. *LMS Journal of Computation and Mathematics*, 5:56–76, August 2002.
9. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL : A theorem-proving environment for higher order logic*. Cambridge University Press, 1993.
10. M. J. C. Gordon, A. R. G. Milner, and C. P. Wadsworth. Edinburgh LCF: A mechanised logic of computation. volume 78 of *LNCS*. Springer-Verlag, 1979.
11. J. Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38(2):162–170, 1995.
12. J. Harrison, 2005. Private communication. See directory `hol_light/Examples/holby.ml` of the HOL Light distribution [14].
13. The HOL-4 Proof Tool. Tool URL <http://hol.sf.net>, 2003.
14. The HOL Light Proof Tool. Tool URL <http://www.cl.cam.ac.uk/users/jrh/hol-light/index.html>, 2005.
15. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
16. A. R. G. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
17. A. R. G. Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. MIT Press, May 1997.
18. George C. Necula and Peter Lee. Proof generation in the Touchstone theorem prover. In David A. McAllester, editor, *Conference on Automated Deduction (CADE)*, volume 1831 of *Lecture Notes in Computer Science*, pages 25–44. Springer, 2000.
19. Peter Sestoft. Moscow ML. <http://www.dina.dk/~sestoft/mosml.html>, 2003.