# Mosquito: an implementation of higher-order logic
## (Rough diamond)

Dominic P. Mulligan*

Computer Laboratory, University of Cambridge

**Abstract.** We present Mosquito: an experimental stateless, pure, largely total LCF-style implementation of higher-order logic using Haskell as a metalanguage. We discuss details of the logic implemented, kernel design and novel proof state and tactic representations.

Mosquito is an experimental LCF-style implementation of higher-order logic (HOL) using Haskell as its metalanguage. The system is under active development. Some simple proofs, both forward and backward, have been performed in the system and various theories are currently under construction. Mosquito's source code may be obtained anonymously from a public Mercurial respository and is developed with Glasgow Haskell Compiler (GHC) 7.6.2.[1]

The motivation behind Mosquito's development is threefold. First, we wish to experiment with the development of an LCF-style implementation of pure HOL in idiomatic Haskell. Whilst Haskell shares many similarities with ML, the typical metalanguage of an LCF-style implementation, it also has many important differences, including: a non-strict evaluation semantics, the lack of ML-style modules, type classes, an institutionalised affinity for purity, widespread use of generic programming, metaprogramming via Template Haskell, type-level programming and other advanced type language features, a novel approach to concurrency via software transactional memory, etc. How these features may be leveraged in a HOL implementation is an interesting question.

Second, we envision the possibility of writing Haskell executables that use Mosquito as a subcomponent for reasoning in HOL. We see a host of possible applications from novel educational software, to programming language semantics animation tools, to writing high-assurance Haskell monitoring software that uses Mosquito to check security or cryptographic properties internally, and so on. Put into context, we see Mosquito eventually being more alike HOL4 [GM93] or HOL Light [Har09]—as an embeddable reasoning subcomponent, or a proof assistant as a *service*—rather than Isabelle [WPN08] or Matita [ARST11], dedicated standalone formal development systems.

Third, we see Mosquito as a prototype implementation of a HOL written with the express purpose of being relatively straightforward to formalise in another

* We wish to thank Anthony Fox, Ohad Kammar, Peter Sewell and Thomas Tuerk for many useful comments and discussions about this work.

[1] https://bitbucket.org/MosquitoProofAssistant/mosquito

proof assistant. Mosquito is written in a stateless, pure and largely total style—folklore bywords for 'easy to formalise'. We have expressly disavowed the use of exceptions to signal failure in favour of error monads. Further, we have adopted a Wiedijk-style stateless kernel [Wie11] in Mosquito to avoid the use of mutable references (or `IORefs` in Haskell parlance), and the associated problem of the `IO` monad proliferating within the types of functions exported by the kernel.

Under Wiedijk's scheme, a constant `C` is equal to a constant `C'` if they share the same name *and* definition, in contrast with other systems such as HOL Light, where an imperative database of existing constant names is maintained in the kernel, preventing a user from redefining a previously defined constant, lest `C` be defined to be both `0` and `1` thus permitting a deduction of $\vdash$ `0` $=$ `1`. In HOL Light and similar systems, constants need only be tested for equality based on their names, as we may safely rely on constants with differing definitions having different names. In Mosquito, defining a constant `C` to be both `0` and `1` is considered 'safe'—at least from a point of view concerned purely with consistency of the logic—as a deduction of $\vdash$ `0` $=$ `1` is not possible. A similar scheme is used to tag type formers, and the associated *abstraction* and *representation* constants, for new types with their defining theorems.

Lastly, a note on the choice of logic implemented in Mosquito. HOL is a simple (described in a handful of inference rules), well understood and widely implemented logic, due to its proven ability in capturing a wide swathe of mathematics and computer science. By implementing HOL in Mosquito we aim to be able to export and import proofs to and from other implementations via OpenTheory [Hur11]. Further, by maintaining compatibility with other HOL systems we may be able to reuse existing automated proof tools, allowing us to achieve a significant degree of automation within Mosquito, until automated proof tools and decision procedures can be written for the system itself, with relatively little outlay of effort [KH12].

Many HOL-like systems currently exist, though two stand out as being particularly related. Wiedijk's Stateless HOL [Wie11] (progenitor of the stateless approach implemented in Mosquito) is a modification of the HOL Light system, wherein the stateful database of existing constant names has been moved outwith the kernel. Empirical testing suggests that Wiedijk's modifications to HOL Light entail only a moderate performance decrease when compared to vanilla HOL Light. Austin and Alexander's HaskHOL kernel [AA13] is a stateless kernel for an extension of HOL with System-F style type quantification and explicit polymorphism. HaskHOL is also implemented in Haskell.

### Mosquito's logic

Mosquito implements a similar higher-order logic to the HOL Light system. Terms $t$, $u$, and so on, are terms of the simply-typed $\lambda$-calculus extended with constants. The full derivation rules for the core logic are presented in Figure 1. Here, $\Gamma$ and $\Delta$ range over arbitrary contexts, $\{\}$ is the empty context, $fv(t)$ is the set of free variables of a term, $t[\vec{\alpha}/\vec{\phi}]$ is a parallel substitution instantiating types

$$\frac{(t =_\alpha u)}{\{\} \vdash t = u}\,(\mathbf{alpha}) \quad \frac{\Gamma \vdash t = u}{\Gamma \vdash u = t}\,(\mathbf{symm}) \quad \frac{\Gamma \vdash t = u \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash t = v}\,(\mathbf{trans})$$

$$\frac{(t : \texttt{Bool})}{\{t\} \vdash t}\,(\mathbf{assm}) \quad \frac{\Gamma \vdash t \quad \Delta \vdash u}{(\Gamma - u) \cup (\Delta - t) \vdash t = u}\,(\mathbf{asym}) \quad \frac{\Gamma \vdash t = u \quad \Delta \vdash u}{\Gamma \cup \Delta \vdash t}\,(\mathbf{eqmp})$$

$$\frac{\Gamma \vdash t = u}{\Gamma \vdash \lambda a{:}\phi.t = \lambda a{:}\phi.u}\,(\mathbf{abs}) \quad \frac{\Gamma \vdash t = u \quad \Delta \vdash v = w}{\Gamma \cup \Delta \vdash t \cdot v = u \cdot w}\,(\mathbf{comb}) \quad \frac{}{\{\} \vdash \lambda a{:}\phi.t \cdot u = t[a/u]}\,(\mathbf{beta})$$

$$\frac{(a \notin fv(t))}{\{\} \vdash \lambda a{:}\phi.(t \cdot a) = t}\,(\mathbf{eta}) \quad \frac{\Gamma \vdash t}{\Gamma[\vec{b}/\vec{u}] \vdash t[\vec{b}/\vec{u}]}\,(\mathbf{inst}) \quad \frac{\Gamma \vdash t}{\Gamma[\vec{\alpha}/\vec{\phi}] \vdash t[\vec{\alpha}/\vec{\phi}]}\,(\mathbf{tyinst})$$

in terms, and $\Gamma[\vec{\alpha}/\vec{\phi}]$ is its pointwise extension to a context (similarly for parallel capture-avoiding substitutions $t[\vec{b}/\vec{u}]$ and $\Gamma[\vec{b}/\vec{u}]$).

Mosquito follows the LCF design philosophy. The kernel exposes several abstract types: `Term`, `Type` and `Theorem`. Modulo bugs in the design of Haskell or the GHC implementation the only way to construct a non-bottom inhabitant of `Theorem` is via appeal to the implementation of the rules in Figure 1.

The kernel exports two primitive *type formers*: `Bool`, the Booleans, and $- \Rightarrow -$, the function space arrow. Each type former is equipped with an *arity*, a natural number detailing the number of types one must supply to the former to construct a new type. Call a fully-applied type former a *type*. The primitive type `Bool` has arity 0, whilst the primitive type former $- \Rightarrow -$ has arity 2. The kernel API ensures one may only construct 'arity correct' types.

The kernel exports a single primitive constant, $- = -$, the equality constant, of type $\alpha \Rightarrow \alpha \Rightarrow \texttt{Bool}$. Equalities in Figure 1, $t = u$, are therefore implemented merely as applications $(= t)u$ with syntactic sugar sprinkled atop. Terms within the kernel are fully type annotated and only type-correct terms may be constructed. Figure 1 should be interpreted as including a series of hidden typing constraints. Under this scheme, the side condition stating $t$ must be a *formula*—a term of type `Bool`—in rule (**assm**) makes sense, and the construction of a term $t = u$ necessarily implies that the terms $t$ and $u$ possess the same type.

Mosquito implements slightly stronger inference rules than the HOL Light kernel. Rule (**alpha**) bakes $\alpha$-equivalence, implemented using 'nominal'-style swappings, into Mosquito's reflexivity rule. In HOL Light a weakened form of (**eta**) is axiomatised outwith the kernel. Similarly, a weakened form of (**beta**) is implemented in the HOL Light kernel. The strengthened versions of $\eta$- and $\beta$-equality are then supplied later, as derived rules. We choose to implement the full versions of these rules directly in Mosquito's kernel.

Mosquito's logic may be extended in three ways. A new constant may be defined as equal to an existing term, modulo restrictions on the free (type-)variables of that term. A new inhabited type may be defined in provable bijection with a subset of an existing type. Lastly, a formula may be asserted freeform as an

axiom. As axioms are 'dangerous' from a consistency point of view, Mosquito, indelibly marks any theorem obtained directly or indirectly from an axiom.

### Tactics and the proof state

Mosquito supports forward proof directly by implementing the rules in Figure 1. For more complex proofs backward proof may be more amenable. Backward proof (as well as a mixture of the two) is supported via *tactics* operating on a *proof state*.

A proof state consists of an incomplete derivation tree—a rose tree with an additional constructor `Hole` corresponding to a proof obligation—coupled with metadata. Each `Hole` has a list of assumptions, the goal to prove at that hole, and a flag signalling whether that goal is *selected* or *unselected*. Branches in the derivation tree are annotated by *justification* functions. These are used to replay the proof in a forward direction once completed, collapsing a derivation tree into a `Theorem`. The proof state API exposes three key functions, where `Inference` is a (monadic) type constructor used to signal failure:

```
mkConjecture :: Term → Inference ProofState
qed :: ProofState → Inference Theorem
act :: ProofState → Tactic → Inference ProofState
```

A new proof state is constructed using `mkConjecture`, taking a formula (failing otherwise) as input and creating a derivation tree consisting of a selected hole. This hole has no assumptions and has the input term as its goal.

A theorem is obtained from a completed backward proof using `qed`. This function fails if the justification functions annotating the derivation tree do not correctly replay the proof in a forward direction, a theorem that fails to match the original conjecture is synthesised, or if the derivation tree remains incomplete.

Backwards proof is progressed via a tactic application using the `act` function. Intuitively, tactics are applied to every selected goal at once, where zero or many goals may be selected at any one time. A tactic applied via `act` must succeed on *every* selected goal for `act` itself to succeed. If a tactic succeeds at a goal it is transformed into a branch in the derivation tree with new selected holes as children, corresponding to the subgoals generated by the tactic. This style of tactic application is similar to a style adopted in Matita [ARST11], with the advantage of permitting sharing of proof chunks across multiple goals.

In systems such as HOL Light, tactics are pieced together via tactic-valued functionals called tacticals. In contrast, we embed the abstract syntax tree of a *proof description language* explicitly as an algebraic data type:

```
data Tactic =
  Apply PreTactic | (⟼) Tactic Tactic | Id | FailWith String |
  Try Tactic | (⊕) Tactic Tactic | Repeat Tactic | SelectGoalsI [Int]
```

The semantics of proof descriptions is given by the `act` function. Assume `p` is an arbitrary proof state. Then:

- A call to `act p (FailWith err)` fails on every selected hole. A call to `act p Id` succeeds on every selected hole, keeping the hole (and thus the proof state `p`) unchanged.
- A call to `act p (Try t)` tries to apply `t` at each selected hole. If `t` fails to transform a given hole, the original hole is successfully reinstated.
- A call to `act p (Repeat t)` tries to applies `t` at each selected hole, and then subsequently at any new selected holes opened. The tactic `t` must apply at least once, with the last proof state successfully modified by `t` returned.
- A call to `act p (t ⤜ t')` first applies `t` to each selected goal. If this does not fail, then `t'` is applied to each selected goal, otherwise the call fails.
- A call to `act p (t ⊕ t')` tries to apply `t` to each selected goal. If this fails, then `t'` is applied to each selected goal instead.
- A call to `act p (SelectGoalsI is)` marks a goal with identification number `i` as being selected if `i ∈ is`, otherwise marks the goal as unselected.

`Apply` lifts a *pretactic* into a tactic. Intuitively, a pretactic is a function that edits a selected hole in the derivation tree. A pretactic is supplied with the assumptions and goal to prove at that hole, and may either choose to fail with an error at that hole or succeed, returning a list of new subgoals which are spliced into the derivation tree as new holes. Pretactics have an entirely 'local' view of the derivation tree—they see only information present at a hole, and are not supplied with information about the rest of the tree.

We discuss `alphaPreTactic` as an example, which closes subgoals $A_1 \dots A_n \vdash_? t = u$ where $t$ and $u$ are $\alpha$-equivalent. The pretactic receives the assumptions $A_1 \dots A_n$ and the goal to prove $t = u$ at a given hole. It ensures the goal is an equality, and the equated terms $t, u$ are $\alpha$-equivalent. If so, the pretactic generates an empty list of new subgoals, coupled with a justification function that makes use of the implementation of (**alpha**) from Figure 1 to 'reverse' its operation.

The `Tactic` embedding may be minimalist, but more complex tactics[2] may be written over this type. For instance, the tactic `repeatN` repeatedly applies a tactic `m` times to a given proof state:

```
repeatN :: Int → Tactic → Tactic
repeatN 0 tactic = Id
repeatN m tactic = tactic ⤜ repeatN (m - 1) tactic
```

More complex tactics 'compile' down into more primitive tactics. This sits in contrast to other systems where complex tactics are created by combining together primitive tactics with tacticals.

Embedding tactics as a data type permits us to write 'metatactics', or tactics that inspect and change other tactics. The `act` function induces equivalences on `Tactic`: say that two tactics `t` and `t'` are equivalent if for any `p` their resulting transformation of `p` using `act` is identical. For example, `Id ⤜ t` is equivalent to `t`, and `Repeat (FailWith err)` is equivalent to `FailWith err`. Following this, we may write a tactic `optimise` which rewrites its input according to these equational laws, producing an equivalent but 'optimised' output.

[2] Henceforth we abuse language and call any `Tactic`-valued Haskell function a tactic.

Several tactics cause divergence of `act`, for example `Repeat Id`. Adopting the position that diverging tactics are buggy, we may write a tactic `replaceBottoms` which inspects its input, replacing diverging tactics with `FailWith err`, where `err` contains information useful in tracking down the source of the divergence.

Lastly, we may examine the small-step evolution of the proof state under the action of a given tactic using a 'debugger', which takes a proof state and a tactic and returns `Inference (ProofState, Maybe Tactic)`. For example, `debug`, called on tactic $\mathtt{t} \rightarrowtail \mathtt{t}'$ and proof state `p`, first applies `t` to `p`. If this succeeds, it successfully returns the new proof state paired with $\mathtt{t}'$, otherwise failing.

**Conclusions**

Mosquito is a stateless implementation of HOL written in Haskell. Some backward and forward proofs of simple conjectures have been carried out in the system using Mosquito's tactic and proof state API. With Mosquito we aim to provide an embeddable library for reasoning in HOL within Haskell applications. We also aim to use Mosquito to experiment with Haskell's novel language and type system features within the context of a proof assistant, and to provide a HOL written with the express intention of being easy to verify correct. We leave this verification as potential future work.

Many other avenues of further work also remain, not least the further development of Mosquito. How quickly Mosquito can check large proofs is an open question, and could be resolved by importing existing proofs written in e.g. HOL Light into the system via OpenTheory. Lastly, how Mosquito's embedding of tactics as a proof description language affects the set of tactics that may be written in the system, for good or for bad, remains to be seen and requires further experimentation.

## References


AA13. Evan Austin and Perry Alexander. Stateless higher-order logic with quantified types. In *ITP*, pages 469–476, 2013.

ARST11. Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69, 2011.

GM93. Michael J. C. Gordon and Tom Melham. *Introduction to HOL, a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.

Har09. John Harrison. HOL Light: an overview. In *TPHOLS*, Lecture Notes in Computer Science, pages 60–66, 2009.

Hur11. Joe Hurd. The OpenTheory standard theory library. In *NFM*, Lecture Notes in Computer Science, pages 177–191, 2011.

KH12. Ramana Kumar and Joe Hurd. Standalone tactics using OpenTheory. In *ITP*, pages 405–411, 2012.

Wie11. Freek Wiedijk. Stateless HOL. In *TYPES*, Electronic Proceedings in Theoretical Computer Science, pages 47–61, 2011.

WPN08. Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In *TPHOLS*, Lecture Notes in Computer Science, pages 33–38, 2008.