HOL Light from the foundations

John Harrison

Amazon Web Services

21st Sep 2023 (09:00–10:30)

The world of interactive theorem provers

A few notable general-purpose theorem provers

There is a (too?) diverse world of interactive theorem provers, e.g

- ACL2
- Agda
- Coq
- HOL (HOL Light, HOL4, ProofPower, HOL Zero, Candle)
- IMPS
- Isabelle
- Lean
- Metamath
- Mizar
- Nuprl
- PVS

A few notable general-purpose theorem provers

There is a (too?) diverse world of interactive theorem provers, e.g

- ACL2
- Agda
- Coq
- HOL (HOL Light, HOL4, ProofPower, HOL Zero, Candle)
- IMPS
- Isabelle
- Lean
- Metamath
- Mizar
- Nuprl
- PVS

See Wiedijk's *The Seventeen Provers of the World* for descriptions of many systems and proofs that $\sqrt{2}$ is irrational.

Different traditions, different characteristics

These systems differ in many core characteristics:

- Underlying foundations
- Implementation and software architecture
- Proof language
- Level of automation
- Pre-proved library size and content

Different traditions, different characteristics

These systems differ in many core characteristics:

- Underlying foundations
- Implementation and software architecture
- Proof language
- Level of automation
- Pre-proved library size and content

See Harrison, Urban and Wiedijk *History of Interactive Theorem Proving* for more on early ITP history and origins.

Foundations

The choice of foundations is a difficult one, sometimes balancing simplicity against flexibility or expressiveness:

Foundations

The choice of foundations is a difficult one, sometimes balancing simplicity against flexibility or expressiveness:

- The ‘traditional’ or ‘standard’ foundation for mathematics is set theory, and some provers do use that
  - Metamath and Isabelle/ZF (standard ZF/ZFC)
  - Mizar (Tarski-Grothendieck set theory)

Foundations

The choice of foundations is a difficult one, sometimes balancing simplicity against flexibility or expressiveness:

- The 'traditional' or 'standard' foundation for mathematics is set theory, and some provers do use that
  - Metamath and Isabelle/ZF (standard ZF/ZFC)
  - Mizar (Tarski-Grothendieck set theory)
- Partly as a result of their computer science interconnections, many provers are based on type theory
  - Simple type theory (HOL family, Candle, Isabelle/HOL)
  - Martin-Löf type theory (Agda, Nuprl)
  - Calculus of inductive constructions (Coq, Matita, Lean)
  - Other typed formalisms (IMPS, PVS, HOL Light QE)
  - Univalent foundations (HoTT, Unimath, rzk, ...)

Foundations

The choice of foundations is a difficult one, sometimes balancing simplicity against flexibility or expressiveness:

- The ‘traditional’ or ‘standard’ foundation for mathematics is set theory, and some provers do use that
  - Metamath and Isabelle/ZF (standard ZF/ZFC)
  - Mizar (Tarski-Grothendieck set theory)
- Partly as a result of their computer science interconnections, many provers are based on type theory
  - Simple type theory (HOL family, Candle, Isabelle/HOL)
  - Martin-Löf type theory (Agda, Nuprl)
  - Calculus of inductive constructions (Coq, Matita, Lean)
  - Other typed formalisms (IMPS, PVS, HOL Light QE)
  - Univalent foundations (HoTT, Unimath, rzk, ...)
- Some are even based on very simple foundations analogous to primitive recursive arithmetic, without explicit quantifiers (ACL2, NQTHM)

Software architecture

If we are trying to use interactive provers to make proofs more reliable, then their own correctness may become an issue. How can we achieve high levels of certainty about their foundations?

Software architecture

If we are trying to use interactive provers to make proofs more reliable, then their own correctness may become an issue. How can we achieve high levels of certainty about their foundations?

The reliability of a theorem prover increases dramatically if its correctness depends only on a small amount of code.

Software architecture

If we are trying to use interactive provers to make proofs more reliable, then their own correctness may become an issue. How can we achieve high levels of certainty about their foundations?

The reliability of a theorem prover increases dramatically if its correctness depends only on a small amount of code.

- de Bruijn approach — generate proofs that can be certified by a simple, separate checker.

Software architecture

If we are trying to use interactive provers to make proofs more reliable, then their own correctness may become an issue. How can we achieve high levels of certainty about their foundations?

The reliability of a theorem prover increases dramatically if its correctness depends only on a small amount of code.

- de Bruijn approach — generate proofs that can be certified by a simple, separate checker.
- LCF approach — reduce all rules to sequences of primitive inferences implemented by a small logical kernel.

Software architecture

If we are trying to use interactive provers to make proofs more reliable, then their own correctness may become an issue. How can we achieve high levels of certainty about their foundations?
The reliability of a theorem prover increases dramatically if its correctness depends only on a small amount of code.

- de Bruijn approach — generate proofs that can be certified by a simple, separate checker.
- LCF approach — reduce all rules to sequences of primitive inferences implemented by a small logical kernel.

The checker or kernel can be much simpler than the prover as a whole, even to the point where it can be verified (Milawa, Candle).

Proof languages

Directly invoking the primitive or derived rules tends to give proofs that are *procedural*.

Proof languages

Directly invoking the primitive or derived rules tends to give proofs that are *procedural*.
A *declarative* style (*what* is to be proved, not *how*) can be nicer:

Proof languages

Directly invoking the primitive or derived rules tends to give proofs that are *procedural*.

A *declarative* style (*what* is to be proved, not *how*) can be nicer:

- Easier to write and understand independent of the prover

Proof languages

Directly invoking the primitive or derived rules tends to give proofs that are *procedural*.

A *declarative* style (*what* is to be proved, not *how*) can be nicer:

- Easier to write and understand independent of the prover
- Easier to modify

Proof languages

Directly invoking the primitive or derived rules tends to give proofs that are *procedural*.

A *declarative* style (*what* is to be proved, not *how*) can be nicer:

- Easier to write and understand independent of the prover
- Easier to modify
- Less tied to the details of the prover, hence more portable

Proof languages

Directly invoking the primitive or derived rules tends to give proofs that are *procedural*.

A *declarative* style (*what* is to be proved, not *how*) can be nicer:

- Easier to write and understand independent of the prover
- Easier to modify
- Less tied to the details of the prover, hence more portable
- However it can also be more verbose and less easy to script.

Proof languages

Directly invoking the primitive or derived rules tends to give proofs that are *procedural*.

A *declarative* style (*what* is to be proved, not *how*) can be nicer:

- Easier to write and understand independent of the prover
- Easier to modify
- Less tied to the details of the prover, hence more portable
- However it can also be more verbose and less easy to script.

Mizar pioneered the declarative style of proof, but key ideas adopted elsewhere, e.g. Isabelle's structured proof language Isar.

Automation

One major obstacle to the wider use of proof assistants is the low level of automation, so it can be a struggle to prove 'obvious' facts. There are some quite powerful automated techniques, e.g.

Automation

One major obstacle to the wider use of proof assistants is the low level of automation, so it can be a struggle to prove 'obvious' facts. There are some quite powerful automated techniques, e.g.

- Pure logic proof search (SAT, FOL, HOL)

Automation

One major obstacle to the wider use of proof assistants is the low level of automation, so it can be a struggle to prove 'obvious' facts. There are some quite powerful automated techniques, e.g.

- Pure logic proof search (SAT, FOL, HOL)
- Decision procedures for numerical theories (linear arithmetic and algebra, SMT).

Automation

One major obstacle to the wider use of proof assistants is the low level of automation, so it can be a struggle to prove 'obvious' facts. There are some quite powerful automated techniques, e.g.

- Pure logic proof search (SAT, FOL, HOL)
- Decision procedures for numerical theories (linear arithmetic and algebra, SMT).
- Quantifier elimination procedures

Automation

One major obstacle to the wider use of proof assistants is the low level of automation, so it can be a struggle to prove 'obvious' facts. There are some quite powerful automated techniques, e.g.

- Pure logic proof search (SAT, FOL, HOL)
- Decision procedures for numerical theories (linear arithmetic and algebra, SMT).
- Quantifier elimination procedures

Many of these have been successfully integrated into proof assistants without compromising their soundness, e.g.

Automation

One major obstacle to the wider use of proof assistants is the low level of automation, so it can be a struggle to prove 'obvious' facts. There are some quite powerful automated techniques, e.g.

- Pure logic proof search (SAT, FOL, HOL)
- Decision procedures for numerical theories (linear arithmetic and algebra, SMT).
- Quantifier elimination procedures

Many of these have been successfully integrated into proof assistants without compromising their soundness, e.g.

- Reimplement algorithms to perform proofs as they proceed

Automation

One major obstacle to the wider use of proof assistants is the low level of automation, so it can be a struggle to prove 'obvious' facts. There are some quite powerful automated techniques, e.g.

- Pure logic proof search (SAT, FOL, HOL)
- Decision procedures for numerical theories (linear arithmetic and algebra, SMT).
- Quantifier elimination procedures

Many of these have been successfully integrated into proof assistants without compromising their soundness, e.g.

- Reimplement algorithms to perform proofs as they proceed
- Have suitable 'certificates' produced by an external tool checked in the inference kernel.

Automation

One major obstacle to the wider use of proof assistants is the low level of automation, so it can be a struggle to prove 'obvious' facts. There are some quite powerful automated techniques, e.g.

- Pure logic proof search (SAT, FOL, HOL)
- Decision procedures for numerical theories (linear arithmetic and algebra, SMT).
- Quantifier elimination procedures

Many of these have been successfully integrated into proof assistants without compromising their soundness, e.g.

- Reimplement algorithms to perform proofs as they proceed
- Have suitable 'certificates' produced by an external tool checked in the inference kernel.
- Extend kernel with verified implementation (*reflection*).

Libraries

- Another serious obstacle is the lack of libraries of ‘basic’ results, meaning that when proving a major theorem one needs constantly to be proving a stream of low-level lemmas.

Libraries

- Another serious obstacle is the lack of libraries of ‘basic’ results, meaning that when proving a major theorem one needs constantly to be proving a stream of low-level lemmas.
- Large formalizations (Odd Order Theorem, Flyspeck) have motivated formalization of ‘foundational’ material as a by-product, making similar efforts easier in future.

Libraries

- Another serious obstacle is the lack of libraries of ‘basic’ results, meaning that when proving a major theorem one needs constantly to be proving a stream of low-level lemmas.
- Large formalizations (Odd Order Theorem, Flyspeck) have motivated formalization of ‘foundational’ material as a by-product, making similar efforts easier in future.
- The earliest large mathematical library is the Mizar Mathematical Library (MML), following the style of mathematical papers with extracted text and references.

Libraries

- Another serious obstacle is the lack of libraries of 'basic' results, meaning that when proving a major theorem one needs constantly to be proving a stream of low-level lemmas.
- Large formalizations (Odd Order Theorem, Flyspeck) have motivated formalization of 'foundational' material as a by-product, making similar efforts easier in future.
- The earliest large mathematical library is the Mizar Mathematical Library (MML), following the style of mathematical papers with extracted text and references.
- Many theorem provers including Coq, HOL Light, Isabelle/HOL and Lean have large and every-expanding libraries.

HOL Light and the LCF approach

HOL Light overview

- HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.

HOL Light overview

- HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.
- An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.

HOL Light overview

- HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.
- An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.
- HOL Light is designed to have a particularly simple and clean logical foundation.

HOL Light overview

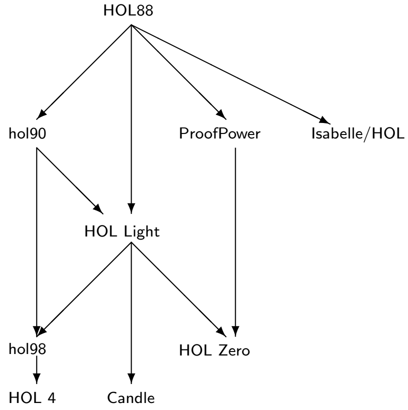
- HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.
- An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.
- HOL Light is designed to have a particularly simple and clean logical foundation.
- Written in Objective CAML (OCaml), a somewhat popular variant of the ML family of languages.

HOL Light overview

- HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.
- An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.
- HOL Light is designed to have a particularly simple and clean logical foundation.
- Written in Objective CAML (OCaml), a somewhat popular variant of the ML family of languages.
- Has been used for proofs in both verification and mathematics, including the Kepler conjecture (Flyspeck project)

The HOL family DAG

There are many HOL provers, all descended from Mike Gordon's original HOL system in the late 1980s.



The LCF approach to theorem proving

- The key ideas of the LCF architecture were invented by Robin Milner and his collaborators in Edinburgh in the 1970s.

The LCF approach to theorem proving

- The key ideas of the LCF architecture were invented by Robin Milner and his collaborators in Edinburgh in the 1970s.
- The original LCF-style prover was for Scott's "Logic of Computable Functions", hence the name, but the approach is not tied to any specific logic.

The LCF approach to theorem proving

- The key ideas of the LCF architecture were invented by Robin Milner and his collaborators in Edinburgh in the 1970s.
- The original LCF-style prover was for Scott's "Logic of Computable Functions", hence the name, but the approach is not tied to any specific logic.
- LCF gives a very attractive mix of *security* and *extensibility/programmability*.

The LCF approach to theorem proving

- The key ideas of the LCF architecture were invented by Robin Milner and his collaborators in Edinburgh in the 1970s.
- The original LCF-style prover was for Scott's "Logic of Computable Functions", hence the name, but the approach is not tied to any specific logic.
- LCF gives a very attractive mix of *security* and *extensibility/programmability*.
- There have been quite a few LCF-style provers for various logics, e.g. HOL, Nuprl, LAMBDA, Isabelle/HOL (and to some extent Coq used the LCF approach).

How an LCF-style prover works

A logical inference rule such as $\Rightarrow$-elimination (*modus ponens*)

$$\frac{\Gamma \vdash p \Rightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

How an LCF-style prover works

A logical inference rule such as $\Rightarrow$-elimination (*modus ponens*)

$$\frac{\Gamma \vdash p \Rightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

becomes a *function*, say `MP : thm->thm->thm` in the metalanguage ML (OCaml in the case of HOL Light)

How an LCF-style prover works

A logical inference rule such as $\Rightarrow$-elimination (*modus ponens*)

$$\frac{\Gamma \vdash p \Rightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

becomes a *function*, say `MP : thm->thm->thm` in the metalanguage ML (OCaml in the case of HOL Light)

For example, if `th1` is the theorem $\vdash p \Rightarrow (q \Rightarrow p)$ and `th2` is $\vdash p$, then `MP th1 th2` gives $\vdash q \Rightarrow p$.

How an LCF-style prover works

A logical inference rule such as $\Rightarrow$-elimination (*modus ponens*)

$$\frac{\Gamma \vdash p \Rightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

becomes a *function*, say `MP : thm->thm->thm` in the metalanguage ML (OCaml in the case of HOL Light)
For example, if `th1` is the theorem $\vdash p \Rightarrow (q \Rightarrow p)$ and `th2` is $\vdash p$, then `MP th1 th2` gives $\vdash q \Rightarrow p$.

- An abstract type of *theorems* can restrict the user to an approved selection of *primitive inference rules* — all theorems must be created with those.

How an LCF-style prover works

A logical inference rule such as $\Rightarrow$-elimination (*modus ponens*)

$$\frac{\Gamma \vdash p \Rightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

becomes a *function*, say `MP : thm->thm->thm` in the metalanguage ML (OCaml in the case of HOL Light)
For example, if `th1` is the theorem $\vdash p \Rightarrow (q \Rightarrow p)$ and `th2` is $\vdash p$, then `MP th1 th2` gives $\vdash q \Rightarrow p$.

- An abstract type of *theorems* can restrict the user to an approved selection of *primitive inference rules* — all theorems must be created with those.
- By layers of programming, much more high-level and convenient *derived inference rules* can be programmed on top.

HOL Light

HOL Light is an extreme case of the LCF approach. The entire logical kernel is 430 lines of code:

- 10 rather simple primitive inference rules
- 2 conservative definitional extension principles
- 3 mathematical axioms (infinity, extensionality, choice)

HOL Light

HOL Light is an extreme case of the LCF approach. The entire logical kernel is 430 lines of code:

- 10 rather simple primitive inference rules
- 2 conservative definitional extension principles
- 3 mathematical axioms (infinity, extensionality, choice)

Arguably, HOL Light is the computer-age descendant of Whitehead and Russell's *Principia Mathematica*:

- The logical basis is simple type theory, which was distilled (Ramsey, Chwistek, Church) from PM's original logic.
- Everything, even arithmetic on numbers, is done from first principles by reduction to the primitive logical basis.

HOL Light and OCaml

- HOL Light is just an OCaml program, so installing HOL Light means installing OCaml and loading HOL Light files into an interactive session

HOL Light and OCaml

- HOL Light is just an OCaml program, so installing HOL Light means installing OCaml and loading HOL Light files into an interactive session
- HOL Light uses camlp5 to make a few modifications to OCaml's usual concrete syntax, which makes things slightly more complicated.

HOL Light and OCaml

- HOL Light is just an OCaml program, so installing HOL Light means installing OCaml and loading HOL Light files into an interactive session
- HOL Light uses camlp5 to make a few modifications to OCaml's usual concrete syntax, which makes things slightly more complicated.
- There are also many similarities between OCaml (the 'metalogic') and the higher-order logic of HOL (the 'object logic'), which can be both illuminating and confusing.

Candle: a verified HOL Light

This project also gives an alternative fully verified version:

`https://cakeml.org/candle`

see Abrahamsson, Myreen, Kumar and Sewell, *Candle: A Verified Implementation of HOL Light*, ITP 2022.

Candle: a verified HOL Light

This project also gives an alternative fully verified version:

$\boxed{\texttt{https://cakeml.org/candle}}$

see Abrahamsson, Myreen, Kumar and Sewell, *Candle: A Verified Implementation of HOL Light*, ITP 2022.

This is based on a completely different software stack, CakeML, a formally verified (in HOL4) ML compiler.

HOL Light applications

Tom Hales and The Kepler conjecture

The Kepler conjecture

The *Kepler conjecture* states that no arrangement of identical balls in ordinary 3-dimensional space has a higher packing density than the obvious 'cannonball' arrangement.

Hales, working with Ferguson, arrived at a proof in 1998:

- 300 pages of mathematics: geometry, measure, graph theory and related combinatorics, ...
- 40,000 lines of supporting computer code: graph enumeration, nonlinear optimization and linear programming.

Hales submitted his proof to *Annals of Mathematics* ...

The response of the reviewers

After a full four years of deliberation, the reviewers returned:

> *"The news from the referees is bad, from my perspective. They have not been able to certify the correctness of the proof, and will not be able to certify it in the future, because they have run out of energy to devote to the problem. This is not what I had hoped for.*
> *Fejes Toth thinks that this situation will occur more and more often in mathematics. He says it is similar to the situation in experimental science — other scientists acting as referees can't certify the correctness of an experiment, they can only subject the paper to consistency checks. He thinks that the mathematical community will have to get used to this state of affairs."*

The response of the reviewers

After a full four years of deliberation, the reviewers returned:

> *"The news from the referees is bad, from my perspective. They have not been able to certify the correctness of the proof, and will not be able to certify it in the future, because they have run out of energy to devote to the problem. This is not what I had hoped for.*
>
> *Fejes Toth thinks that this situation will occur more and more often in mathematics. He says it is similar to the situation in experimental science — other scientists acting as referees can't certify the correctness of an experiment, they can only subject the paper to consistency checks. He thinks that the mathematical community will have to get used to this state of affairs."*

Dissatisfied with the state of affairs, Hales initiated a project called *Flyspeck* ('Formal Proof of the Kepler Conjecture') to completely formalize the proof.

Flyspeck: structure of the formal proof

A large team effort led by Hales brought Flyspeck to completion:

Flyspeck: structure of the formal proof

A large team effort led by Hales brought Flyspeck to completion:

- All the ordinary mathematics has been formalized in HOL Light: Euclidean geometry, measure theory, *hypermaps*, *fans*, results on packings (work of many people).

Flyspeck: structure of the formal proof

A large team effort led by Hales brought Flyspeck to completion:

- All the ordinary mathematics has been formalized in HOL Light: Euclidean geometry, measure theory, *hypermaps*, *fans*, results on packings (work of many people).
- Linear programming part has been proved using highly optimized derived rules for LP certification in HOL Light (Alexey Solovyev, earlier work by Steven Obua)

Flyspeck: structure of the formal proof

A large team effort led by Hales brought Flyspeck to completion:

- All the ordinary mathematics has been formalized in HOL Light: Euclidean geometry, measure theory, *hypermaps*, *fans*, results on packings (work of many people).
- Linear programming part has been proved using highly optimized derived rules for LP certification in HOL Light (Alexey Solovyev, earlier work by Steven Obua)
- Nonlinear optimization parts have been verified in using highly optimized derived rules for interval reasoning in HOL Light (Alexey Solovyev).

Flyspeck: structure of the formal proof

A large team effort led by Hales brought Flyspeck to completion:

- All the ordinary mathematics has been formalized in HOL Light: Euclidean geometry, measure theory, *hypermaps*, *fans*, results on packings (work of many people).
- Linear programming part has been proved using highly optimized derived rules for LP certification in HOL Light (Alexey Solovyev, earlier work by Steven Obua)
- Nonlinear optimization parts have been verified in using highly optimized derived rules for interval reasoning in HOL Light (Alexey Solovyev).
- The graph enumeration process has been verified in Isabelle/HOL via ML code generation (Tobias Nipkow).

Flyspeck and the future of mathematics

"In truth, my motivations for the project are far more complex than a simple hope of removing residual doubt from the minds of few referees. Indeed, I see formal methods as fundamental to the long-term growth of mathematics. (Hales, The Kepler Conjecture*)*

s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: machine code with state-of-the-art performance

s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: machine code with state-of-the-art performance
- Secure: all code is written in "constant-time" style

s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: machine code with state-of-the-art performance
- Secure: all code is written in "constant-time" style
- Correct: every function is formally verified in HOL Light

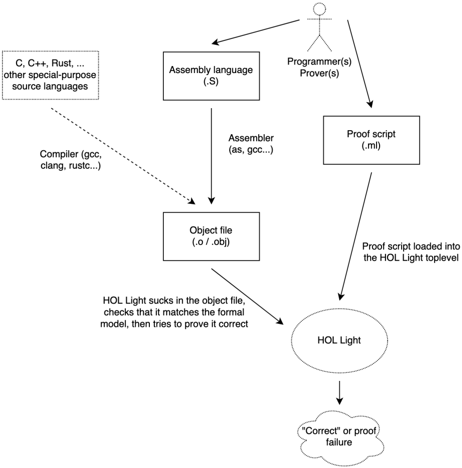
s2n-bignum

An open-source library of bignum arithmetic operations designed for cryptographic applications.

- Efficient: machine code with state-of-the-art performance
- Secure: all code is written in "constant-time" style
- Correct: every function is formally verified in HOL Light

`https://github.com/awslabs/s2n-bignum`

Coding and verification flow



Verifying the actual code

Formalization of code as byte sequence derived from the object file:

```
define_assert_from_elf "bignum_montmul_p256_mc"
  "arm/p256/bignum_montmul_p256.o"
```

Verifying the actual code

Formalization of code as byte sequence derived from the object file:

```
define_assert_from_elf "bignum_montmul_p256_mc"
  "arm/p256/bignum_montmul_p256.o"
```

Automatically re-check when code and/or proof changes:

```
p256/%.correct: proofs/%.ml p256/%.o ; .....
```

Verifying the actual code

Formalization of code as byte sequence derived from the object file:

```
define_assert_from_elf "bignum_montmul_p256_mc"
  "arm/p256/bignum_montmul_p256.o"
```

Automatically re-check when code and/or proof changes:

```
p256/%.correct: proofs/%.ml p256/%.o ; .....
```

Run in continuous integration on any github pull request

Installation and OCaml basics

Installing OCaml

There are standard packages for OCaml, but it can be fiddly getting an appropriate version of OCaml and camlp5, so we recommend Opam:

`https://opam.ocaml.org/doc/Install.html`

Then this should give suitable prerequisites for HOL Light:

```
opam init
eval `opam env`
opam switch create 4.14.0
eval `opam env`
opam pin add camlp5 8.00.04
opam install num camlp5 camlp-streams ocamlfind
```

Installing HOL Light

Clone the git repo:

```
git clone https://github.com/jrh13/hol-light.git
```

Installing HOL Light

Clone the git repo:

```
git clone https://github.com/jrh13/hol-light.git
```

Go into the HOL Light directory and make the camlp5 extension:

```
cd ./hol-light
make
```

Installing HOL Light

Clone the git repo:

```
git clone https://github.com/jrh13/hol-light.git
```

Go into the HOL Light directory and make the camlp5 extension:

```
cd ./hol-light
make
```

Start OCaml and load the HOL Light root file

```
ocaml
#use "hol.ml";;
```

Wait 1-2 minutes for everything to load

The OCaml toplevel

When using HOL Light, you are in the top-level read-eval-print loop of OCaml, a strongly typed functional programming language.

- OCaml presents the prompt '#'
- Enter phrases terminated by *double* semicolon ';;' for evaluation

The OCaml toplevel

When using HOL Light, you are in the top-level read-eval-print loop of OCaml, a strongly typed functional programming language.

- OCaml presents the prompt '#'
- Enter phrases terminated by *double* semicolon ';;' for evaluation

The user enters

```
# 2 + 2;;
```

The OCaml toplevel

When using HOL Light, you are in the top-level read-eval-print loop of OCaml, a strongly typed functional programming language.

- OCaml presents the prompt '#'
- Enter phrases terminated by *double* semicolon ';;' for evaluation

The user enters

```
# 2 + 2;;
```

and OCaml responds with

```
val it : int = 4
#
```

It not only returns the *value* (4) but also infers the type (int) and binds it to a name (it).

OCaml bindings

We can now use the name ‘it’ to stand for that expression:

```
# it * it;;
val it : int = 16
```

OCaml bindings

We can now use the name ‘it’ to stand for that expression:

```
# it * it;;
val it : int = 16
```

We can also choose our own names for bindings using ‘let *name* $=$ *expression*’, with multiple parallel bindings separated by ‘and’:

```
# let a = 2 and b = 3;;
val a : int = 2
val b : int = 3
# let c = a - b;;
val c : int = -1
```

OCaml bindings

We can now use the name 'it' to stand for that expression:

```
# it * it;;
val it : int = 16
```

We can also choose our own names for bindings using 'let *name* = *expression*', with multiple parallel bindings separated by 'and':

```
# let a = 2 and b = 3;;
val a : int = 2
val b : int = 3
# let c = a - b;;
val c : int = -1
```

or make bindings *local* to an expression using 'in':

```
# let d = a / 2 in d + 6;;
val it : int = 7
# d;;
Error: Unbound value d
```

Basic OCaml datatypes

A few basic built-in datatypes:

- Integers (int), which we've already seen, written in the usual way. Note that these are machine integers with limited range.

Basic OCaml datatypes

A few basic built-in datatypes:

- Integers (int), which we've already seen, written in the usual way. Note that these are machine integers with limited range.
- Floating-point values (float) written with the decimal point like '1.0'. The operations on FP numbers are different, '+.' etc.

Basic OCaml datatypes

A few basic built-in datatypes:

- Integers (int), which we've already seen, written in the usual way. Note that these are machine integers with limited range.
- Floating-point values (float) written with the decimal point like '1.0'. The operations on FP numbers are different, '`+.`' etc.
- Booleans (bool), with elements `false` and `true` and operations like infix '`&&`' and '`||`'

Basic OCaml datatypes

A few basic built-in datatypes:

- Integers (int), which we've already seen, written in the usual way. Note that these are machine integers with limited range.
- Floating-point values (float) written with the decimal point like '1.0'. The operations on FP numbers are different, '`+.`' etc.
- Booleans (bool), with elements `false` and `true` and operations like infix '`&&`' and '`||`'
- Strings (string) written in `"Double quotes"` with '`^`' as infix concatenation.

Pairs and lists

OCaml has two especially important structured datatypes, though the user can define more (and HOL Light defines its own for logical concepts);

- Pairs, written with an infix ‘,’ (the parentheses are only needed to establish precedence)

  ```
  # 1,2;;
  val it : int * int = (1, 2)
  ```

Pairs and lists

OCaml has two especially important structured datatypes, though the user can define more (and HOL Light defines its own for logical concepts);

- Pairs, written with an infix ',' (the parentheses are only needed to establish precedence)

  ```
  # 1,2;;
  val it : int * int = (1, 2)
  ```

- Lists, written with *semicolon* as separator, and `::` as 'cons':

  ```
  # 1::2::[3;4];;
  val it : int list = [1; 2; 3; 4]
  ```

Pairs and lists

OCaml has two especially important structured datatypes, though the user can define more (and HOL Light defines its own for logical concepts);

- Pairs, written with an infix ',' (the parentheses are only needed to establish precedence)

  ```
  # 1,2;;
  val it : int * int = (1, 2)
  ```

- Lists, written with *semicolon* as separator, and `::` as 'cons':

  ```
  # 1::2::[3;4];;
  val it : int list = [1; 2; 3; 4]
  ```

Structured types can be nested in arbitrary ways (lists of pairs of lists etc.) and OCaml automatically keeps track of the types.

OCaml functions

One can define *functions* in OCaml using either of the following more or less equivalent forms:

- An explicit 'lambda' written '`fun v -> e`', e.g.

```
# let square = fun x -> x * x;;
val square : int -> int = <fun>
```

OCaml functions

One can define *functions* in OCaml using either of the following more or less equivalent forms:

- An explicit 'lambda' written '`fun v -> e`', e.g.

```
# let square = fun x -> x * x;;
val square : int -> int = <fun>
```

- An ordinary let-binding with parameters

```
# let square x = x * x;;
val square : int -> int = <fun>
```

OCaml functions

One can define *functions* in OCaml using either of the following more or less equivalent forms:

- An explicit 'lambda' written '`fun v -> e`', e.g.

  ```
  # let square = fun x -> x * x;;
  val square : int -> int = <fun>
  ```

- An ordinary let-binding with parameters

  ```
  # let square x = x * x;;
  val square : int -> int = <fun>
  ```

Functions are applied just by juxtaposition; parentheses are only needed to establish precedence

```
# square 12 + 1;;
val it : int = 145
# square (12 + 1);;
val it : int = 169
```

Recursion and pattern-matching

Function definitions can be recursive with the `rec` keyword, and since OCaml is primarily a functional language, this is a major control flow mechanism.

- The factorial function can be defined as

```
# let rec fact n = if n <= 0 then 1 else n * fact(n - 1);;
val fact : int -> int = <fun>
# fact 12;;
val it : int = 479001600
```

- The length of a list can be determined as follows; note the use of pattern-matching 'match ... with' clauses:

```
# let rec length l =
  match l with
    [] -> 0
  | h::t -> 1 + length t;;
val length : 'a list -> int = <fun>
# length [1;2;3];;
val it : int = 3
```

Currying

OCaml allows function types to be nested, so one can implement multiple-argument functions as functions returning functions ('currying').

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# let suc = add 1;;
val suc : int -> int = <fun>
# suc 2;;
val it : int = 3
```

Currying

OCaml allows function types to be nested, so one can implement multiple-argument functions as functions returning functions ('currying').

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# let suc = add 1;;
val suc : int -> int = <fun>
# suc 2;;
val it : int = 3
```

Alternatively one can explicitly use a paired argument:

```
# let add(x,y) = x + y;;
val add : int * int -> int = <fun>
# add(1,3);;
val it : int = 4
```

Polymorphism

OCaml infers 'most general' types for functions according to an elegant polymorphic type system, with 'type variables' used to signify generality.

```
# let identity x = x;;
val identity : 'a -> 'a = <fun>
```

Polymorphism

OCaml infers 'most general' types for functions according to an elegant polymorphic type system, with 'type variables' used to signify generality.

```
# let identity x = x;;
val identity : 'a -> 'a = <fun>
```

Such a function can be applied to any specific instance (or a more complex polymorphic type)

```
# identity 1;;
val it : int = 1
# identity false;;
val it : bool = false
```

HOL Light basics

Basic logical entities in OCaml

There are three key OCaml datatypes used to represent logical entities in HOL:

- Higher-order logic *types*, `hol_type`. You can conveniently create them using specially parsed backquotes with colon:

```
# `:bool`;;
val it : hol_type = `:bool`
```

Basic logical entities in OCaml

There are three key OCaml datatypes used to represent logical entities in HOL:

- Higher-order logic *types*, `hol_type`. You can conveniently create them using specially parsed backquotes with colon:

```
# `:bool`;;
val it : hol_type = `:bool`
```

- HOL terms, `term`, which can also be conveniently created via special parsing support

```
# `1 + 2`;;
val it : term = `1 + 2`
```

Basic logical entities in OCaml

There are three key OCaml datatypes used to represent logical entities in HOL:

- Higher-order logic *types*, `hol_type`. You can conveniently create them using specially parsed backquotes with colon:

```
# `:bool`;;
val it : hol_type = `:bool`
```

- HOL terms, `term`, which can also be conveniently created via special parsing support

```
# `1 + 2`;;
val it : term = `1 + 2`
```

- HOL theorems, which cannot be just created arbitrarily but must be *proved*, e.g. the pre-existing theorem that addition is commutative.

```
# ADD_SYM;;
val it : thm = |- !m n. m + n = n + m
```

Abstract type encapsulation

All the three core logical datatypes are effectively abstract data types, so how you can form them is *restricted* to ensure logical coherence

- You can only create HOL types that have been declared

```
# `:int triple`;;
Exception: Failure "Unparsed input following type".
```

Abstract type encapsulation

All the three core logical datatypes are effectively abstract data types, so how you can form them is *restricted* to ensure logical coherence

- You can only create HOL types that have been declared

```
# `:int triple`;;
Exception: Failure "Unparsed input following type".
```

- You can only create well-typed HOL terms; here we try to add 1 and 'false' (the Booleans are written as F and T in HOL):

```
# `1 + F`;;
Exception:
Failure
 "typechecking error (initial type assignment): F has type bool, it cannot
used with type num".
```

Abstract type encapsulation

All the three core logical datatypes are effectively abstract data types, so how you can form them is *restricted* to ensure logical coherence

- You can only create HOL types that have been declared

```
# `:int triple`;;
Exception: Failure "Unparsed input following type".
```

- You can only create well-typed HOL terms; here we try to add 1 and 'false' (the Booleans are written as F and T in HOL):

```
# `1 + F`;;
Exception:
Failure
 "typechecking error (initial type assignment): F has type bool, it cannot
used with type num".
```

- Theorems can only be created (ultimately) by applying a small number of primitive rules