HOL Light — from foundations to applications

John Harrison

Intel Corporation

18th May 2015 (08:30–10:00)

Summary of talk

- The world of interactive theorem provers
- HOL Light and the LCF approach
- HOL Light in formal verification and pure mathematics
- Installation and OCaml basics
- The HOL Logic in OCaml

The world of interactive theorem provers

A few notable general-purpose theorem provers

There is a diverse (perhaps too diverse?) world of proof assistants, with these being just a few:

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

A few notable general-purpose theorem provers

There is a diverse (perhaps too diverse?) world of proof assistants, with these being just a few:

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

See Freek Wiedijk's book *The Seventeen Provers of the World* (Springer-Verlag lecture notes in computer science volume 3600) for descriptions of many systems and proofs that $\sqrt{2}$ is irrational.

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
  - HOL family and Isabelle/HOL (simple type theory)
  - Martin-Löf type theory (Agda, Nuprl)
  - Calculus of inductive constructions (Coq)
  - Other typed formalisms (IMPS, PVS)

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
  - HOL family and Isabelle/HOL (simple type theory)
  - Martin-Löf type theory (Agda, Nuprl)
  - Calculus of inductive constructions (Coq)
  - Other typed formalisms (IMPS, PVS)
- Some are even based on very simple foundations analogous to primitive recursive arithmetic, without explicit quantifiers (ACL2, NQTHM)

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
  - HOL family and Isabelle/HOL (simple type theory)
  - Martin-Löf type theory (Agda, Nuprl)
  - Calculus of inductive constructions (Coq)
  - Other typed formalisms (IMPS, PVS)
- Some are even based on very simple foundations analogous to primitive recursive arithmetic, without explicit quantifiers (ACL2, NQTHM)
- There is now interest in a new foundational approach, homotopy type theory, with experimental implementations.

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.

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.
There have even recently been papers about versions of Milawa (a simplified ACL2) and HOL Light verified right down to machine code.

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. Recently, several other declarative proof languages have been developed, as well as declarative shells round existing systems like HOL and Isabelle.

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.
- Sometimes flashy or exciting theorems (Brouwer fixed-point theorem, the Picard theorems) aren’t as useful as less showy ones (the change of variables formula for integrals etc.)

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.
- Sometimes flashy or exciting theorems (Brouwer fixed-point theorem, the Picard theorems) aren't as useful as less showy ones (the change of variables formula for integrals etc.)
- 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.
- Sometimes flashy or exciting theorems (Brouwer fixed-point theorem, the Picard theorems) aren't as useful as less showy ones (the change of variables formula for integrals etc.)
- 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, still perhaps the largest 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.
- Sometimes flashy or exciting theorems (Brouwer fixed-point theorem, the Picard theorems) aren't as useful as less showy ones (the change of variables formula for integrals etc.)
- 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, still perhaps the largest is the Mizar Mathematical Library (MML), following the style of mathematical papers with extracted text and references.
- Many theorem provers including Coq, HOL Light and Isabelle/HOL (including the 'archive of formal proofs') also have large and every-expanding mathematical 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

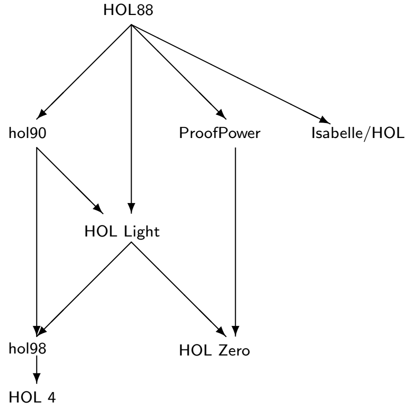
- HOL Light is a member of the HOL family of provers, descended from Mike Gordon's original HOL system developed in the 80s.
- An LCF-style proof checker for classical higher-order logic built on top of (polymorphic) simply-typed λ-calculus.
- HOL Light is designed to have a particularly simple and clean 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 floating-point algorithm verifications at Intel and the verification of Hales's proof of the Kepler conjecture (Flyspeck).

The HOL family DAG

There are many HOL provers, of which HOL Light is just one, 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 (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 (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 (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*)

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

becomes a *function*, say `MP : thm->thm->thm` in the metalanguage (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

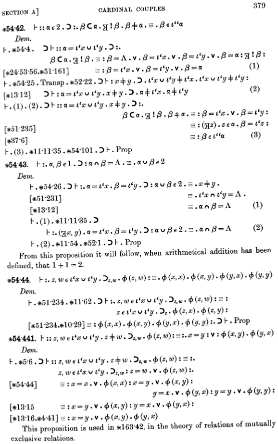
HOL Light is an extreme case of the LCF approach. The entire logical kernel is 430 lines of code:

- 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.

A formal proof from 1910



$*54{\cdot}42$. $\vdash :: \alpha \epsilon 2 \,.\, \supset :.\, \beta \subset \alpha \,.\, \exists ! \beta \,.\, \beta \neq \alpha \,.\, \equiv \,.\, \beta \epsilon \iota“\alpha$

Dem.

$\vdash \,.\, *54{\cdot}4 \,.\, \supset \vdash :: \alpha = \iota‘x \cup \iota‘y \,.\, \supset :.$

$\beta \subset \alpha \,.\, \exists ! \beta \,.\, \equiv : \beta = \Lambda \,.\, \vee \,.\, \beta = \iota‘x \,.\, \vee \,.\, \beta = \iota‘y \,.\, \vee \,.\, \beta = \alpha : \exists ! \beta :$

$[*24{\cdot}53{\cdot}56 \,.\, *51{\cdot}161]$ $\equiv : \beta = \iota‘x \,.\, \vee \,.\, \beta = \iota‘y \,.\, \vee \,.\, \beta = \alpha$ (1)

$\vdash \,.\, *54{\cdot}25 \,.\, \text{Transp} \,.\, *52{\cdot}22 \,.\, \supset \vdash : x \neq y \,.\, \supset \,.\, \iota‘x \cup \iota‘y \neq \iota‘x \,.\, \iota‘x \cup \iota‘y \neq \iota‘y :$

$[*13{\cdot}12]$ $\supset \vdash : \alpha = \iota‘x \cup \iota‘y \,.\, x \neq y \,.\, \supset \,.\, \alpha \neq \iota‘x \,.\, \alpha \neq \iota‘y$ (2)

$\vdash \,.\, (1) \,.\, (2) \,.\, \supset \vdash :: \alpha = \iota‘x \cup \iota‘y \,.\, x \neq y \,.\, \supset :.$

$\beta \subset \alpha \,.\, \exists ! \beta \,.\, \beta \neq \alpha \,.\, \equiv : \beta = \iota‘x \,.\, \vee \,.\, \beta = \iota‘y :$

$[*51{\cdot}235]$ $\equiv : (\exists z) \,.\, z \epsilon \alpha \,.\, \beta = \iota‘z :$

$[*37{\cdot}6]$ $\equiv : \beta \epsilon \iota“\alpha$ (3)

$\vdash \,.\, (3) \,.\, *11{\cdot}11{\cdot}35 \,.\, *54{\cdot}101 \,.\, \supset \vdash \,.\,$ Prop

$*54{\cdot}43$. $\vdash :.\, \alpha, \beta \epsilon 1 \,.\, \supset : \alpha \cap \beta = \Lambda \,.\, \equiv \,.\, \alpha \cup \beta \epsilon 2$

Dem.

$\vdash \,.\, *54{\cdot}26 \,.\, \supset \vdash :.\, \alpha = \iota‘x \,.\, \beta = \iota‘y \,.\, \supset : \alpha \cup \beta \epsilon 2 \,.\, \equiv \,.\, x \neq y \,.$

$[*51{\cdot}231]$ $\equiv \,.\, \iota‘x \cap \iota‘y = \Lambda \,.$

$[*13{\cdot}12]$ $\equiv \,.\, \alpha \cap \beta = \Lambda$ (1)

$\vdash \,.\, (1) \,.\, *11{\cdot}11{\cdot}35 \,.\, \supset$

$\vdash :.\, (\exists x, y) \,.\, \alpha = \iota‘x \,.\, \beta = \iota‘y \,.\, \supset : \alpha \cup \beta \epsilon 2 \,.\, \equiv \,.\, \alpha \cap \beta = \Lambda$ (2)

$\vdash \,.\, (2) \,.\, *11{\cdot}54 \,.\, *52{\cdot}1 \,.\, \supset \vdash \,.\,$ Prop

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.

$*54{\cdot}44$. $\vdash :.\, z, w \epsilon \iota‘x \cup \iota‘y \,.\, \supset_{z,w} \,.\, \phi(z, w) : \equiv \,.\, \phi(x, x) \,.\, \phi(x, y) \,.\, \phi(y, x) \,.\, \phi(y, y)$

Dem.

$\vdash \,.\, *51{\cdot}234 \,.\, *11{\cdot}62 \,.\, \supset \vdash :.\, z, w \epsilon \iota‘x \cup \iota‘y \,.\, \supset_{z,w} \,.\, \phi(z, w) : \equiv :$

$z \epsilon \iota‘x \cup \iota‘y \,.\, \supset_z \,.\, \phi(z, x) \,.\, \phi(z, y) :$

$[*51{\cdot}234 \,.\, *10{\cdot}29]$ $\equiv : \phi(x, x) \,.\, \phi(x, y) \,.\, \phi(y, x) \,.\, \phi(y, y) :.\, \supset \vdash \,.\,$ Prop

$*54{\cdot}441$. $\vdash :: z, w \epsilon \iota‘x \cup \iota‘y \,.\, z \neq w \,.\, \supset_{z,w} \,.\, \phi(z, w) : \equiv :.\, x = y : \vee : \phi(x, y) \,.\, \phi(y, x)$

Dem.

$\vdash \,.\, *5{\cdot}6 \,.\, \supset \vdash :: z, w \epsilon \iota‘x \cup \iota‘y \,.\, z \neq w \,.\, \supset_{z,w} \,.\, \phi(z, w) : \equiv :.$

$z, w \epsilon \iota‘x \cup \iota‘y \,.\, \supset_{z,w} : z = w \,.\, \vee \,.\, \phi(z, w) :.$

$[*54{\cdot}44]$ $\equiv : x = x \,.\, \vee \,.\, \phi(x, x) : x = y \,.\, \vee \,.\, \phi(x, y) :$

$y = x \,.\, \vee \,.\, \phi(y, x) : y = y \,.\, \vee \,.\, \phi(y, y) :$

$[*13{\cdot}15$ $\equiv : x = y \,.\, \vee \,.\, \phi(x, y) : y = x \,.\, \vee \,.\, \phi(y, x) :$

$[*13{\cdot}16 \,.\, *4{\cdot}41]$ $\equiv : x = y \,.\, \vee \,.\, \phi(x, y) \,.\, \phi(y, x)$

This proposition is used in $*163{\cdot}42$, in the theory of relations of mutually exclusive relations.

This is p379 of Whitehead and Russell's *Principia Mathematica*.

Zooming in . . .

$*54{\cdot}43$. $\vdash :. \alpha, \beta \in 1 . \supset : \alpha \cap \beta = \Lambda . \equiv . \alpha \cup \beta \in 2$

Dem.

$$\vdash . *54{\cdot}26 . \supset \vdash :. \alpha = \iota`x . \beta = \iota`y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y .$$

$$[*51{\cdot}231] \qquad \equiv . \iota`x \cap \iota`y = \Lambda .$$

$$[*13{\cdot}12] \qquad \equiv . \alpha \cap \beta = \Lambda \qquad (1)$$

$$\vdash . (1) . *11{\cdot}11{\cdot}35 . \supset$$

$$\vdash :. (\exists x, y) . \alpha = \iota`x . \beta = \iota`y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda \qquad (2)$$

$$\vdash . (2) . *11{\cdot}54 . *52{\cdot}1 . \supset \vdash . \text{Prop}$$

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.

A formal proof from 2010

```
let PNT = prove
 (`((\n. &(CARD {p | prime p /\ p <= n}) / (&n / log(&n)))
    ---> &1) sequentially`,
  REWRITE_TAC[PNT_PARTIAL_SUMMATION] THEN
  REWRITE_TAC[SUM_PARTIAL_PRE] THEN
  REWRITE_TAC[GSYM REAL_OF_NUM_ADD; SUB_REFL; CONJUNCT1 LE] THEN
  SUBGOAL_THEN `{p | prime p /\ p = 0} = {}` SUBST1_TAC THENL
   [REWRITE_TAC[EXTENSION; IN_ELIM_THM; NOT_IN_EMPTY] THEN
    MESON_TAC[PRIME_IMP_NZ];
    ALL_TAC] THEN
  REWRITE_TAC[SUM_CLAUSES; REAL_MUL_RZERO; REAL_SUB_RZERO] THEN
  MATCH_MP_TAC REALLIM_TRANSFORM_EVENTUALLY THEN
  EXISTS_TAC
   `\n. ((&n + &1) / log(&n + &1) *
         sum {p | prime p /\ p <= n} (\p. log(&p) / &p) -
         sum (1..n)
         (\k. sum {p | prime p /\ p <= k} (\p. log(&p) / &p) *
              ((&k + &1) / log(&k + &1) - &k / log(&k)))) / (&n / log(&n))` THEN
  CONJ_TAC THENL
   [REWRITE_TAC[EVENTUALLY_SEQUENTIALLY] THEN EXISTS_TAC `1` THEN SIMP_TAC[];
    ALL_TAC] THEN
  MATCH_MP_TAC REALLIM_TRANSFORM THEN
  EXISTS_TAC
   `\n. ((&n + &1) / log(&n + &1) * log(&n) -
         sum (1..n)
         (\k. log(&k) * ((&k + &1) / log(&k + &1) - &k / log(&k)))) /
        (&n / log(&n))` THEN
  REWRITE_TAC[] THEN CONJ_TAC THENL
   [REWRITE_TAC[REAL_ARITH
     `(a * x - s) / b - (a * x' - s') / b:real =
      ((s' - s) - (x' - x) * a) / b`] THEN
    REWRITE_TAC[GSYM SUM_SUB_NUMSEG; GSYM REAL_SUB_RDISTRIB] THEN
    REWRITE_TAC[REAL_OF_NUM_ADD] THEN
    MATCH_MP_TAC SUM_PARTIAL_LIMIT_ALT THEN
```

Zooming in ...

At least the theorems are more substantial:

```
let PNT = prove
 (`((\n. &(CARD {p | prime p /\ p <= n}) / (&n / log(&n)))
    ---> &1) sequentially`,
  REWRITE_TAC[PNT_PARTIAL_SUMMATION] THEN
  REWRITE_TAC[SUM_PARTIAL_PRE] THEN
  REWRITE_TAC[GSYM REAL_OF_NUM_ADD; SUB_REFL; CONJUNCT1 LE] THEN
```

Zooming in . . .

At least the theorems are more substantial:

```
let PNT = prove
 (`((\n. &(CARD {p | prime p /\ p <= n}) / (&n / log(&n)))
    ---> &1) sequentially`,
  REWRITE_TAC[PNT_PARTIAL_SUMMATION] THEN
  REWRITE_TAC[SUM_PARTIAL_PRE] THEN
  REWRITE_TAC[GSYM REAL_OF_NUM_ADD; SUB_REFL; CONJUNCT1 LE] THEN
```

Though whether formal proofs have become more digestible to the non-expert is perhaps questionable . . .

HOL Light in formal verification and mathematics

Intel's diverse activities

Intel is best known as a hardware company, and hardware is still the core of the company's business. However this entails much more:

- Microcode
- Firmware
- Protocols
- Software

Intel's diverse activities

Intel is best known as a hardware company, and hardware is still the core of the company's business. However this entails much more:

- Microcode
- Firmware
- Protocols
- Software

If the Intel® Software and Services Group (SSG) were split off as a separate company, it would be in the top 10 software companies worldwide.

A diversity of verification problems

This gives rise to a corresponding diversity of verification problems, and of verification solutions.

- Propositional tautology/equivalence checking (FEV)
- Symbolic simulation
- Symbolic trajectory evaluation (STE)
- Temporal logic model checking
- Combined decision procedures (SMT)
- First order automated theorem proving
- Interactive theorem proving

Most of these techniques (trading automation for generality / efficiency) are in active use at Intel.

A spectrum of formal techniques

Traditionally, formal verification has been focused on complete proofs of functional correctness.

But recently there have been notable successes elsewhere for 'semi-formal' methods involving abstraction or more limited property checking.

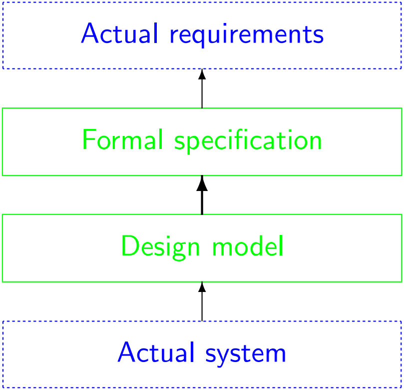
- Airbus A380 avionics
- Microsoft SLAM/SDV

One can also consider applying theorem proving technology to support testing or other traditional validation methods like path coverage.

These are all areas of interest at Intel.

Models and their validation

We have the usual concerns about validating our specs, but also need to pay attention to the correspondence between our models and physical reality.



Physical problems

Chips can suffer from physical problems, usually due to overheating or particle bombardment (‘soft errors’).

Physical problems

Chips can suffer from physical problems, usually due to overheating or particle bombardment ('soft errors').

- In 1978, Intel encountered problems with 'soft errors' in some of its DRAM chips.

Physical problems

Chips can suffer from physical problems, usually due to overheating or particle bombardment ('soft errors').

- In 1978, Intel encountered problems with 'soft errors' in some of its DRAM chips.
- The cause turned out to be alpha particle emission from the packaging.

Physical problems

Chips can suffer from physical problems, usually due to overheating or particle bombardment ('soft errors').

- In 1978, Intel encountered problems with 'soft errors' in some of its DRAM chips.
- The cause turned out to be alpha particle emission from the packaging.
- The factory producing the ceramic packaging was on the Green River in Colorado, downstream from the tailings of an old uranium mine.

Physical problems

Chips can suffer from physical problems, usually due to overheating or particle bombardment ('soft errors').

- In 1978, Intel encountered problems with 'soft errors' in some of its DRAM chips.
- The cause turned out to be alpha particle emission from the packaging.
- The factory producing the ceramic packaging was on the Green River in Colorado, downstream from the tailings of an old uranium mine.

However, these are rare and apparently well controlled by existing engineering best practice.

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors
- Very rarely encountered, but was hit by a mathematician doing research in number theory.

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors
- Very rarely encountered, but was hit by a mathematician doing research in number theory.
- Intel eventually set aside US $475 million to cover the costs.

The FDIV bug

Formal methods are more useful for avoiding design errors such as the infamous FDIV bug:

- Error in the floating-point division (FDIV) instruction on some early Intel®Pentium® processors
- Very rarely encountered, but was hit by a mathematician doing research in number theory.
- Intel eventually set aside US $475 million to cover the costs.

This did at least considerably improve investment in formal verification.

Some HOL Light verifications

We have formally verified correctness of various floating-point algorithms using HOL Light:

- Division and square root (Marstein-style, using fused multiply-add to do Newton-Raphson or power series approximation with delicate final rounding).
- Transcendental functions like *log* and *sin* (table-driven algorithms using range reduction and a core polynomial approximations).

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 birth of Flyspeck

Hales's proof was eventually published, and no significant error has been found in it. Nevertheless, the verdict is disappointingly lacking in clarity and finality.

As a result of this experience, the journal changed its editorial policy on computer proof so that it will no longer even try to check the correctness of computer code.

Dissatisfied with this state of affairs, Hales initiated a project called *Flyspeck* to completely formalize the proof.

Flyspeck

Flyspeck = 'Formal Proof of the Kepler Conjecture'.

> *"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*)*

In parallel, Hales has simplified the informal proof using ideas from Marchal, significantly cutting down on the formalization work.

Flyspeck: current status

A large team effort led by Hales brought Flyspeck to completion on 10th August 2014:

Flyspeck: current status

A large team effort led by Hales brought Flyspeck to completion on 10th August 2014:

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

Flyspeck: current status

A large team effort led by Hales brought Flyspeck to completion on 10th August 2014:

- All the ordinary mathematics has been formalized in HOL Light: Euclidean geometry, measure theory, *hypermaps*, *fans*, results on packings.
- The graph enumeration process has been verified (and improved in the process) by Tobias Nipkow in Isabelle/HOL.

Flyspeck: current status

A large team effort led by Hales brought Flyspeck to completion on 10th August 2014:

- All the ordinary mathematics has been formalized in HOL Light: Euclidean geometry, measure theory, *hypermaps*, *fans*, results on packings.
- The graph enumeration process has been verified (and improved in the process) by Tobias Nipkow in Isabelle/HOL.
- A highly optimized way of formally proving the linear programming part in HOL Light has been developed by Alexey Solovyev, following earlier work by Steven Obua.

Flyspeck: current status

A large team effort led by Hales brought Flyspeck to completion on 10th August 2014:

- All the ordinary mathematics has been formalized in HOL Light: Euclidean geometry, measure theory, *hypermaps*, *fans*, results on packings.
- The graph enumeration process has been verified (and improved in the process) by Tobias Nipkow in Isabelle/HOL.
- A highly optimized way of formally proving the linear programming part in HOL Light has been developed by Alexey Solovyev, following earlier work by Steven Obua.
- A method has been developed by Alexey Solovyev to prove all the nonlinear optimization results, running in many parallel sessions of HOL Light.

OCaml basics

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.

Installation basics

The difficulty of installation varies with operating system. This page is the main guide:

```
https://code.google.com/p/hol-light/source/browse/trunk/README
```

Installation basics

The difficulty of installation varies with operating system. This page is the main guide:

```
https://code.google.com/p/hol-light/source/browse/trunk/README
```

There is a debian package for HOL Light (thanks to Hendrik Tews), so for debian and derivatives like Ubuntu you can simply do

```
sudo apt-get install hol-light
```

then start it up with the following (it takes a minute or so to load everything in)

```
hol-light
```

Installation basics

The difficulty of installation varies with operating system. This page is the main guide:

`https://code.google.com/p/hol-light/source/browse/trunk/README`

There is a debian package for HOL Light (thanks to Hendrik Tews), so for debian and derivatives like Ubuntu you can simply do

```
sudo apt-get install hol-light
```

then start it up with the following (it takes a minute or so to load everything in)

```
hol-light
```

For other OSs you will probably need to install OCaml, camlp5 and then HOL Light itself separately.

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

HOL types

In general, a HOL type is either

- A polymorphic type variable

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

HOL types

In general, a HOL type is either

- A polymorphic type variable

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

- A compound type built up from basic types using a type operator, like the function space ->, lists or pairs

```
# `:num->bool list`;;
val it : hol_type = `:num->(bool)list`
```

HOL types

In general, a HOL type is either

- A polymorphic type variable

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

- A compound type built up from basic types using a type operator, like the function space ->, lists or pairs

```
# `:num->bool list`;;
val it : hol_type = `:num->(bool)list`
```

- Note that certain basic types like `bool` are considered as nullary type operators.

HOL types

In general, a HOL type is either

- A polymorphic type variable

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

- A compound type built up from basic types using a type operator, like the function space ->, lists or pairs

```
# `:num->bool list`;;
val it : hol_type = `:num->(bool)list`
```

- Note that certain basic types like `bool` are considered as nullary type operators.

The type system is very closely analogous to that of OCaml itself, and HOL's parser even uses similar algorithms to assign most general polymorphic types.

HOL terms

There are only four basic kinds of HOL term:

HOL terms

There are only four basic kinds of HOL term:

- Variables, with a specific type

```
# `p:bool`;;
val it : term = `p`
```

HOL terms

There are only four basic kinds of HOL term:

- Variables, with a specific type

```
# `p:bool`;;
val it : term = `p`
```

- Constants, again with a specific type that HOL Light will usually infer, though it supports some degree of constant overloading

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

HOL terms

There are only four basic kinds of HOL term:

- Variables, with a specific type

```
# `p:bool`;;
val it : term = `p`
```

- Constants, again with a specific type that HOL Light will usually infer, though it supports some degree of constant overloading

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

- Applications, written with juxtaposition (this is the successor function applied to 0):

```
# `SUC 0`;;
val it : term = `SUC 0`
```

HOL terms

There are only four basic kinds of HOL term:

- Variables, with a specific type

```
# `p:bool`;;
val it : term = `p`
```

- Constants, again with a specific type that HOL Light will usually infer, though it supports some degree of constant overloading

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

- Applications, written with juxtaposition (this is the successor function applied to 0):

```
# `SUC 0`;;
val it : term = `SUC 0`
```

- Abstractions or lambdas, written with a backslash

```
# `\x. x + 1`;;
val it : term = `\x. x + 1`
```

HOL Light primitive rules (1)

$$\frac{}{\vdash t = t}\ \texttt{REFL}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \texttt{TRANS}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \texttt{MK\_COMB}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.\ s) = (\lambda x.\ t)}\ \texttt{ABS}$$

$$\frac{}{\vdash (\lambda x.\ t)x = t}\ \texttt{BETA}$$

HOL Light primitive rules (2)

$$\frac{}{\{p\} \vdash p} \; \texttt{ASSUME}$$

$$\frac{\Gamma \vdash p = q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q} \; \texttt{EQ\_MP}$$

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p = q} \; \texttt{DEDUCT\_ANTISYM\_RULE}$$

$$\frac{\Gamma[x_1, \ldots, x_n] \vdash p[x_1, \ldots, x_n]}{\Gamma[t_1, \ldots, t_n] \vdash p[t_1, \ldots, t_n]} \; \texttt{INST}$$

$$\frac{\Gamma[\alpha_1, \ldots, \alpha_n] \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma[\gamma_1, \ldots, \gamma_n] \vdash p[\gamma_1, \ldots, \gamma_n]} \; \texttt{INST\_TYPE}$$

HOL's logical connectives

The usual logical connectives are given ASCII renderings:

$\bot$	`F`	Falsity
$\top$	`T`	Truth
$\neg$	`~`	Not
$\wedge$	`/\`	And
$\vee$	`\/`	Or
$\Rightarrow$	`==>`	Implies ('if ... then ...')
$\Leftrightarrow$	`<=>`	Iff ('... if and only if ...')
$\forall$	`!`	For all
$\exists$	`?`	There exists
$\exists!$	`?!`	There exists a unique

The definitions of the logical connectives

HOL Light is so foundational that even all the basic logical connectives are *defined* in terms of equality:

$$
\begin{array}{rcl}
\top & = & (\lambda p.\ p) = (\lambda p.\ p) \\
\wedge & = & \lambda p.\ \lambda q.\ (\lambda f.\ f\ p\ q) = (\lambda f.\ f\ \top\ \top) \\
\Rightarrow & = & \lambda p.\ \lambda q.\ p \wedge q = p \\
\forall & = & \lambda P.\ P = \lambda x.\ \top \\
\exists & = & \lambda P.\ \forall q.\ (\forall x.\ P(x) \Rightarrow q) \Rightarrow q \\
\vee & = & \lambda p.\ \lambda q.\ \forall r.\ (p \Rightarrow r) \Rightarrow (q \Rightarrow r) \Rightarrow r \\
\bot & = & \forall p.\ p \\
\neg & = & \lambda p.\ p \Rightarrow \bot \\
\exists! & = & \lambda P.\ \exists P \wedge \forall x.\ \forall y.\ P\ x \wedge P\ y \Rightarrow (x = y)
\end{array}
$$

The usual properties of the connectives are *derived* from the primitive rules.

Basic syntax functions

HOL Light provides many convenient function for manipulating the basic logical entities, e.g.

Basic syntax functions

HOL Light provides many convenient function for manipulating the basic logical entities, e.g.

- `type_of` to get the (HOL!) type of a term

```
# type_of `1`;;
val it : hol_type = `:num`
```

Basic syntax functions

HOL Light provides many convenient function for manipulating the basic logical entities, e.g.

- `type_of` to get the (HOL!) type of a term

```
# type_of `1`;;
val it : hol_type = `:num`
```

- Destructor functions `dest_var`, `dest_const`, `dest_comb` and `dest_abs` to break down terms of various kinds

```
# dest_comb `SUC 0`;;
val it : term * term = (`SUC`, `0`)
```

Basic syntax functions

HOL Light provides many convenient function for manipulating the basic logical entities, e.g.

- type_of to get the (HOL!) type of a term

```
# type_of `1`;;
val it : hol_type = `:num`
```

- Destructor functions dest_var, dest_const, dest_comb and dest_abs to break down terms of various kinds

```
# dest_comb `SUC 0`;;
val it : term * term = (`SUC`, `0`)
```

- Corresponding constructors mk_var, mk_const, mk_comb and mk_abs

```
# mk_var("p",`:bool`);;
val it : term = `p`
```

Basic syntax functions

HOL Light provides many convenient function for manipulating the basic logical entities, e.g.

- type_of to get the (HOL!) type of a term

```
# type_of `1`;;
val it : hol_type = `:num`
```

- Destructor functions dest_var, dest_const, dest_comb and dest_abs to break down terms of various kinds

```
# dest_comb `SUC 0`;;
val it : term * term = (`SUC`, `0`)
```

- Corresponding constructors mk_var, mk_const, mk_comb and mk_abs

```
# mk_var("p",`:bool`);;
val it : term = `p`
```

- frees to get the free variables in a term

```
# frees `x + y + 1`;;
val it : term list = [`x`; `y`]
```

Representing more complex terms

All the expressions in logic and mathematics are ultimately expressed using just those four basic terms, and one can explore how it is done using the destructor functions

- Binary logical connectives are just curried functions of the appropriate type:

```
# dest_comb `p /\ q`;;
val it : term * term = (`(/\) p`, `q`)
```

- Quantifiers are higher-order functions applied to an abstraction

```
# dest_comb `!x. x < x + 1`;;
val it : term * term = (`(!)`, `\x. x < x + 1`)
```

Getting help

Note that one can also get help on any predefined HOL Light functions using the `help` function, e.g.

```
# help "mk_abs";;
```

Getting help

Note that one can also get help on any predefined HOL Light functions using the `help` function, e.g.

```
# help "mk_abs";;
```

There is also a full Reference manual with the same information.

HOL Light — from foundations to applications

John Harrison

Intel Corporation

19th May 2015 (10:30–12:00)

Summary of talk

- Basic and derived definitional principles
- Basic mathematical theories in HOL Light
- More advanced automation
- Tactic proofs
- A tour of the library

Basic and derived definitional principles

Basic principle of constant definition

The only primitive constant for the logic itself is equality `=` with polymorphic type $\alpha \to \alpha \to \texttt{bool}$.

Basic principle of constant definition

The only primitive constant for the logic itself is equality `=` with polymorphic type $\alpha \to \alpha \to \texttt{bool}$.
Later we add the Hilbert $\varepsilon : (\alpha \to \texttt{bool}) \to \alpha$ yielding the Axiom of Choice.

Basic principle of constant definition

The only primitive constant for the logic itself is equality = with polymorphic type $\alpha \to \alpha \to \mathtt{bool}$.

Later we add the Hilbert $\varepsilon : (\alpha \to \mathtt{bool}) \to \alpha$ yielding the Axiom of Choice.

All other constants are introduced using `new_basic_definition`, the rule of constant definition: given a term t (closed, and with some restrictions on type variables) and an unused constant name c, we can define c and get the new theorem

$$\vdash c = t$$

Basic principle of constant definition

The only primitive constant for the logic itself is equality $=$ with polymorphic type $\alpha \to \alpha \to \texttt{bool}$.

Later we add the Hilbert $\varepsilon : (\alpha \to \texttt{bool}) \to \alpha$ yielding the Axiom of Choice.

All other constants are introduced using `new_basic_definition`, the rule of constant definition: given a term t (closed, and with some restrictions on type variables) and an unused constant name c, we can define c and get the new theorem

$$\vdash c = t$$

This is an object-level definitional principle, in that c is a constant, not some meta-level abbreviation. It is easy to see that this is conservative, and in particular consistency-preserving.

Basic principle of type definition

The only primitive type constructors for the logic itself are `bool` (booleans) and `fun` (function space).

Basic principle of type definition

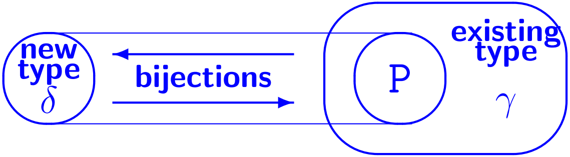
The only primitive type constructors for the logic itself are `bool` (booleans) and `fun` (function space).
Later we add an infinite type `ind` (individuals) to assert the axiom of infinity.

Basic principle of type definition

The only primitive type constructors for the logic itself are `bool` (booleans) and `fun` (function space).

Later we add an infinite type `ind` (individuals) to assert the axiom of infinity.

All other types are introduced by `new_basic_type_definition`, the rule of type definition, to be in bijection with any nonempty subset of an existing type.



Again, this is conservative and consistency-preserving.

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.

One of the innovations of Gordon's original HOL work was to extend this 'correct-by-construction' approach to the definitions of new concepts, which works very nicely in a general framework like HOL, so:

- All proofs are done by primitive inferences
- All new types are defined not postulated.

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.

One of the innovations of Gordon's original HOL work was to extend this 'correct-by-construction' approach to the definitions of new concepts, which works very nicely in a general framework like HOL, so:

- All proofs are done by primitive inferences
- All new types are defined not postulated.

This is the standard approach in mathematics, even if most of the time people don't bother about it (e.g. the construction of the real numbers as Dedekind cuts or whatever).

HOL as a definitional framework

While Edinburgh LCF required theorems to be proved via the primitive inference rules, it was usual to assert axioms to give the definitions required, and it was quite easy to assert inconsistent axioms.

One of the innovations of Gordon's original HOL work was to extend this 'correct-by-construction' approach to the definitions of new concepts, which works very nicely in a general framework like HOL, so:

- All proofs are done by primitive inferences
- All new types are defined not postulated.

This is the standard approach in mathematics, even if most of the time people don't bother about it (e.g. the construction of the real numbers as Dedekind cuts or whatever).

Just using axioms was compared by Russell to theft in place of honest toil.

Convenient higher-level definitional principles

However, part of the motivation for just axiomatizing definitions is that it's often very convenient to use much higher-level principles, e.g.

Convenient higher-level definitional principles

However, part of the motivation for just axiomatizing definitions is that it's often very convenient to use much higher-level principles, e.g.

- Inductive definitions of sets and predicates
- Definition of inductive types (trees, lists etc.)
- Definition of primitive recursive functions over such types
- Definition of general recursive functions using wellfounded orderings

Convenient higher-level definitional principles

However, part of the motivation for just axiomatizing definitions is that it's often very convenient to use much higher-level principles, e.g.

- Inductive definitions of sets and predicates
- Definition of inductive types (trees, lists etc.)
- Definition of primitive recursive functions over such types
- Definition of general recursive functions using wellfounded orderings

Many other theorem provers build such principles in as primitive, and very often get them wrong . . .

Convenient higher-level definitional principles

However, part of the motivation for just axiomatizing definitions is that it's often very convenient to use much higher-level principles, e.g.

- Inductive definitions of sets and predicates
- Definition of inductive types (trees, lists etc.)
- Definition of primitive recursive functions over such types
- Definition of general recursive functions using wellfounded orderings

Many other theorem provers build such principles in as primitive, and very often get them wrong ...
HOL Light supports all these and more using safely *derived* definitional principles.

Inductively defined relations

The `new_inductive_definition` function automates inductive definitions, using a Knaster-Tarski type derivation under the surface. It can cope with infinitary definitions, parameters, and user-defined monotone operators.

Inductively defined relations

The `new_inductive_definition` function automates inductive definitions, using a Knaster-Tarski type derivation under the surface. It can cope with infinitary definitions, parameters, and user-defined monotone operators.

```
# new_inductive_definition ‘E(0) /\ (!n. E(n) ==> E(n + 2))‘;;
val it : thm * thm * thm =
  (|- E 0 /\ (!n. E n ==> E (n + 2)),
   |- !E’. E’ 0 /\ (!n. E’ n ==> E’ (n + 2)) ==> (!a. E a ==> E’ a),
   |- !a. E a <=> a = 0 \/ (?n. a = n + 2 /\ E n))
```

Inductively defined relations

The `new_inductive_definition` function automates inductive definitions, using a Knaster-Tarski type derivation under the surface. It can cope with infinitary definitions, parameters, and user-defined monotone operators.

```
# new_inductive_definition `E(0) /\ (!n. E(n) ==> E(n + 2))`;;
val it : thm * thm * thm =
  (|- E 0 /\ (!n. E n ==> E (n + 2)),
   |- !E'. E' 0 /\ (!n. E' n ==> E' (n + 2)) ==> (!a. E a ==> E' a),
   |- !a. E a <=> a = 0 \/ (?n. a = n + 2 /\ E n))
```

The function returns a triple of theorems:

- A 'rule' theorem (the inductively defined predicate is closed under the rules)
- An 'induction' or minimality theorem (the inductively defined predicate is the least such)
- A 'cases' theorem that each element arises by virtue of one of the rules.

Inductive/recursive datatypes

These are analogous to the concrete datatypes of OCaml and similar languages. Examples include natural numbers, lists and trees.

Inductive/recursive datatypes

These are analogous to the concrete datatypes of OCaml and similar languages. Examples include natural numbers, lists and trees.

HOL Light's `define_type` rule can handle nested constructors and mutual recursion. For example, a simple type for binary trees with natural numbers at the leaves:

Inductive/recursive datatypes

These are analogous to the concrete datatypes of OCaml and similar languages. Examples include natural numbers, lists and trees.

HOL Light's `define_type` rule can handle nested constructors and mutual recursion. For example, a simple type for binary trees with natural numbers at the leaves:

```
# let btree_INDUCT,btree_RECURSION = define_type
  "btree = Leaf num | Branch btree btree";;
```

Inductive/recursive datatypes

These are analogous to the concrete datatypes of OCaml and similar languages. Examples include natural numbers, lists and trees.

HOL Light's `define_type` rule can handle nested constructors and mutual recursion. For example, a simple type for binary trees with natural numbers at the leaves:

```
# let btree_INDUCT,btree_RECURSION = define_type
  "btree = Leaf num | Branch btree btree";;
```

The rule returns a pair of theorem, one justifying 'structural induction' over the type:

```
val btree_INDUCT : thm =
 |- !P. (!a. P (Leaf a)) /\ (!a0 a1. P a0 /\ P a1 ==> P (Branch a0 a1))
        ==> (!x. P x)
```

Inductive/recursive datatypes

These are analogous to the concrete datatypes of OCaml and similar languages. Examples include natural numbers, lists and trees.

HOL Light's `define_type` rule can handle nested constructors and mutual recursion. For example, a simple type for binary trees with natural numbers at the leaves:

```
# let btree_INDUCT,btree_RECURSION = define_type
  "btree = Leaf num | Branch btree btree";;
```

The rule returns a pair of theorem, one justifying 'structural induction' over the type:

```
val btree_INDUCT : thm =
 |- !P. (!a. P (Leaf a)) /\ (!a0 a1. P a0 /\ P a1 ==> P (Branch a0 a1))
         ==> (!x. P x)
```

and the other justifying definition by primitive recursion

```
val btree_RECURSION : thm =
|- !f0 f1.
         ?fn. (!a. fn (Leaf a) = f0 a) /\
              (!a0 a1. fn (Branch a0 a1) = f1 a0 a1 (fn a0) (fn a1))
```

Recursive functions

HOL Light can automatically use the recursion theorems produced by `define_type` to justify primitive recursive theorems.

Recursive functions

HOL Light can automatically use the recursion theorems produced by `define_type` to justify primitive recursive theorems.

Can also handle general recursive definitions, and in simple cases can find an appropriate wellfounded ordering automatically:

Recursive functions

HOL Light can automatically use the recursion theorems produced by `define_type` to justify primitive recursive theorems.

Can also handle general recursive definitions, and in simple cases can find an appropriate wellfounded ordering automatically:

```
let fib = define
   `fib 0 = 1 /\
    fib 1 = 1 /\
    fib (n + 2) = fib(n) + fib(n + 1)`;;
val fib : thm =
|- fib 0 = 1 /\ fib 1 = 1 /\ fib (n + 2) = fib n + fib (n + 1)
```

Recursive functions

HOL Light can automatically use the recursion theorems produced by `define_type` to justify primitive recursive theorems.
Can also handle general recursive definitions, and in simple cases can find an appropriate wellfounded ordering automatically:

```
let fib = define
   `fib 0 = 1 /\
    fib 1 = 1 /\
    fib (n + 2) = fib(n) + fib(n + 1)`;;
val fib : thm =
|- fib 0 = 1 /\ fib 1 = 1 /\ fib (n + 2) = fib n + fib (n + 1)
```

Some tail-recursive cases can be justified even without an ordering:

```
define `collatz(n) = if n <= 1 then n
                     else if EVEN(n) then collatz(n DIV 2)
                     else collatz(3 * n + 1)`;;
```

Basic mathematical theories in HOL Light

Cartesian products and pairs

We define a Cartesian product constructor written as infix '#' (*not* '* as in OCaml).
This takes two types α and β and gives us the Cartesian product $\alpha \times \beta$.

Cartesian products and pairs

We define a Cartesian product constructor written as infix '#' (*not* '* as in OCaml).
This takes two types α and β and gives us the Cartesian product $\alpha \times \beta$.
As with OCaml, the pairing function is an infix comma, and parentheses are not needed except to establish precedence.

```
# type_of `1,2`;;
val it : hol_type = `:num#num`
```

The projections are FST and SND.

Natural numbers

The axiom of infinity (`INFINITY_AX`) asserts that there is a function from the type of 'individuals' to itself that is *injective* but not *surjective* (Dedekind's definition of infinity)

Natural numbers

The axiom of infinity (`INFINITY_AX`) asserts that there is a function from the type of 'individuals' to itself that is *injective* but not *surjective* (Dedekind's definition of infinity)

This means the type of individuals is big enough to hold the natural numbers, and they are carved out as an inductively defined predicate to use in a type definition.

Natural numbers

The axiom of infinity (`INFINITY_AX`) asserts that there is a function from the type of 'individuals' to itself that is *injective* but not *surjective* (Dedekind's definition of infinity)

This means the type of individuals is big enough to hold the natural numbers, and they are carved out as an inductively defined predicate to use in a type definition.

This gives the type of natural numbers `:num`, a function `SUC` (the image under the bijection of the function postulated by `INFINITY_AX`) and a constant zero (some value not in the range of `SUC`).

Natural numbers

The axiom of infinity (`INFINITY_AX`) asserts that there is a function from the type of 'individuals' to itself that is *injective* but not *surjective* (Dedekind's definition of infinity)

This means the type of individuals is big enough to hold the natural numbers, and they are carved out as an inductively defined predicate to use in a type definition.

This gives the type of natural numbers `:num`, a function `SUC` (the image under the bijection of the function postulated by `INFINITY_AX`) and a constant zero (some value not in the range of `SUC`).

All the usual arithmetical operations are defined and the usual properties proved, making heavy use of definition by recursion and proof by recursion, e.g. the primitive recursive definition of addition:

```
val it : thm = |- (!n. 0 + n = n) /\ (!m n. SUC m + n = SUC (m + n))
```

Natural number constants

The 'constants' $0, 1, 2, 3, 4, \ldots$ are not in fact constants, but prettyprinted forms of composite terms. We use two basic constants for the functions $n \mapsto 2n$ and $n \mapsto 2n + 1$:

```
BIT0 = |- BIT0 n = n + n
```

```
BIT1 = |- BIT1 n = SUC(n + n)
```

Natural number constants

The 'constants' $0, 1, 2, 3, 4, \ldots$ are not in fact constants, but prettyprinted forms of composite terms. We use two basic constants for the functions $n \mapsto 2n$ and $n \mapsto 2n + 1$:

```
BIT0 = |- BIT0 n = n + n
```

```
BIT1 = |- BIT1 n = SUC(n + n)
```

These are used to encode numbers in a binary notation, e,g. 6 as

```
BIT0 (BIT1 (BIT1 _0)
```

Natural number constants

The 'constants' $0, 1, 2, 3, 4, \ldots$ are not in fact constants, but prettyprinted forms of composite terms. We use two basic constants for the functions $n \mapsto 2n$ and $n \mapsto 2n + 1$:

```
BIT0 = |- BIT0 n = n + n

BIT1 = |- BIT1 n = SUC(n + n)
```

These are used to encode numbers in a binary notation, e,g. 6 as

```
BIT0 (BIT1 (BIT1 _0)
```

An outer identity constant `NUMERAL` is applied, which among other things avoids confusing cases where one number is a subterm of another one. So for example:

```
# dest_comb `14`;;
val it : term * term = (`NUMERAL`, `BIT0 (BIT1 (BIT1 (BIT1 _0)))`)
```

Natural number arithmetic

Most arithmetic operations in this representation can be evaluated by applying theorems as rewrite rules

```
ARITH_ADD =
  |- (!m n. NUMERAL m + NUMERAL n = NUMERAL (m + n)) /\
     _0 + _0 = _0 /\
     (!n. _0 + BIT0 n = BIT0 n) /\
     (!n. _0 + BIT1 n = BIT1 n) /\
     (!n. BIT0 n + _0 = BIT0 n) /\
     (!n. BIT1 n + _0 = BIT1 n) /\
     (!m n. BIT0 m + BIT0 n = BIT0 (m + n)) /\
     (!m n. BIT0 m + BIT1 n = BIT1 (m + n)) /\
     (!m n. BIT1 m + BIT0 n = BIT1 (m + n)) /\
     (!m n. BIT1 m + BIT1 n = BIT0 (SUC (m + n)))

ARITH_SUC =
  |- (!n. SUC (NUMERAL n) = NUMERAL (SUC n)) /\
     SUC _0 = BIT1 _0 /\
     (!n. SUC (BIT0 n) = BIT1 n) /\
     (!n. SUC (BIT1 n) = BIT0 (SUC n))
```

Optimized derived rules can do most arithmetic fairly efficiently, way slower than machine arithmetic or bignums, but fast enough for most purposes.

Real numbers (1)

We say a function $x : \mathbb{N} \to \mathbb{N}$ (i.e. a sequence of natural numbers) is *nearly additive* if there is a bound B with

$$\forall m,\ n.\ |x_{m+n} - (x_m + x_n)| \leq B$$

Real numbers (1)

We say a function $x : \mathbb{N} \to \mathbb{N}$ (i.e. a sequence of natural numbers) is *nearly additive* if there is a bound B with

$$\forall m,\ n.\ |x_{m+n} - (x_m + x_n)| \leq B$$

This turns out to be equivalent to being 'nearly multiplicative', i.e. for some B:

$$\forall m, n.\ |m x_n - n x_m| \leq B(m+n)$$

Real numbers (1)

We say a function $x : \mathbb{N} \to \mathbb{N}$ (i.e. a sequence of natural numbers) is *nearly additive* if there is a bound B with

$$\forall m,\ n.\ |x_{m+n} - (x_m + x_n)| \leq B$$

This turns out to be equivalent to being 'nearly multiplicative', i.e. for some B:

$$\forall m, n.\ |m x_n - n x_m| \leq B(m+n)$$

Intuitively, it may help to think of x_n/n converging to a real number. We can turn this round and use it as a *definition* of (nonnegative) real numbers.

Real numbers (2)

Nonnegative reals are defined as equivalence classes of nearly multiplicative sequences. The operations are very easy, for two sequences x_n and y_n:

Real numbers (2)

Nonnegative reals are defined as equivalence classes of nearly multiplicative sequences. The operations are very easy, for two sequences x_n and y_n:

- Addition is just pointwise addition $n \mapsto x_n + y_n$

Real numbers (2)

Nonnegative reals are defined as equivalence classes of nearly multiplicative sequences. The operations are very easy, for two sequences x_n and y_n:

- Addition is just pointwise addition $n \mapsto x_n + y_n$
- Multiplication is actually function composition $n \mapsto x_{y_n}$.

Real numbers (2)

Nonnegative reals are defined as equivalence classes of nearly multiplicative sequences. The operations are very easy, for two sequences x_n and y_n:

- Addition is just pointwise addition $n \mapsto x_n + y_n$
- Multiplication is actually function composition $n \mapsto x_{y_n}$.

Taking appropriate equivalence classes of pairs (thinking of (x, y) as $x - y$) gives the positive and negative reals.

Real numbers (2)

Nonnegative reals are defined as equivalence classes of nearly multiplicative sequences. The operations are very easy, for two sequences x_n and y_n:

- Addition is just pointwise addition $n \mapsto x_n + y_n$
- Multiplication is actually function composition $n \mapsto x_{y_n}$.

Taking appropriate equivalence classes of pairs (thinking of (x, y) as $x - y$) gives the positive and negative reals.

We prove the ‘complete ordered field’ properties and thereafter never look back inside the actual definition, so the precise definition used doesn’t really matter.

Sets

In some sense sets in HOL are trivial: we don't have a special type operator for sets over a type α, but just use predicates, i.e. functions of type $\alpha \to \textsf{bool}$.

Sets

In some sense sets in HOL are trivial: we don't have a special type operator for sets over a type α, but just use predicates, i.e. functions of type $\alpha \to \text{bool}$.
But for familiarity of notation we define a membership relation `IN`

```
|- !P x. x IN P <=> P x
```

Sets

In some sense sets in HOL are trivial: we don't have a special type operator for sets over a type α, but just use predicates, i.e. functions of type $\alpha \to \text{bool}$.
But for familiarity of notation we define a membership relation `IN`

```
|- !P x. x IN P <=> P x
```

as well as a derived syntax (printed in the familiar way by the prettyprinter) for set comprehensions $\{f(x) \mid P(x)\}$ for 'the set of $f(x)$ such that $P(x)$', and the usual set operations, e.g.

```
|- s UNION t = {x | x IN s \/ x IN t}
```

More advanced automation

More automated derived rules

HOL Light does have quite a few quite highly automated derived rules that can prove non-trival properties in the right domains completely automatically (and with the usual proof generation).

- Tautology checker
- First-order automation (MESON, Holyhammer)
- Basic set theory
- Algebra via Gröbner bases
- Linear arithmetic
- ...

More automated derived rules

HOL Light does have quite a few quite highly automated derived rules that can prove non-trival properties in the right domains completely automatically (and with the usual proof generation).

- Tautology checker
- First-order automation (MESON, Holyhammer)
- Basic set theory
- Algebra via Gröbner bases
- Linear arithmetic
- ...

To become productive at formal proof, it's worth appreciating what can and cannot be done by these automated methods.

Tautology checker

You can prove basic propositional tautologies with `TAUT`

```
TAUT `p /\ q <=> p <=> q <=> p \/ q`;;
```

Tautology checker

You can prove basic propositional tautologies with `TAUT`

```
TAUT `p /\ q <=> p <=> q <=> p \/ q`;;
```

This uses a fairly naive algorithm, but Hasan Amjad has developed far more efficient tautology checkers (in the `Minisat` directory) based on the use of external SAT solvers Minisat or zchaff:

Tautology checker

You can prove basic propositional tautologies with `TAUT`

```
TAUT `p /\ q <=> p <=> q <=> p \/ q`;;
```

This uses a fairly naive algorithm, but Hasan Amjad has developed far more efficient tautology checkers (in the `Minisat` directory) based on the use of external SAT solvers Minisat or zchaff:

- Convert the problem to standard format and call the SAT solver
- Use the proof trace returned to generate a HOL Light proof.

Tautology checker

You can prove basic propositional tautologies with `TAUT`

```
TAUT `p /\ q <=> p <=> q <=> p \/ q`;;
```

This uses a fairly naive algorithm, but Hasan Amjad has developed far more efficient tautology checkers (in the `Minisat` directory) based on the use of external SAT solvers Minisat or zchaff:

- Convert the problem to standard format and call the SAT solver
- Use the proof trace returned to generate a HOL Light proof.

The HOL Light proof generation time is not usually much more than the existing search time for the SAT solver.

First-order automation

HOL Light has a simple first-order prover `MESON` based on model elimination, which can dispose of much purely first-order reasoning, e.g.

First-order automation

HOL Light has a simple first-order prover `MESON` based on model elimination, which can dispose of much purely first-order reasoning, e.g.

```
MESON[]
  `(!x y z. P x y /\ P y z ==> P x z) /\
   (!x y z. Q x y /\ Q y z ==> Q x z) /\
   (!x y. P x y ==> P y x) /\
   (!x y. P x y \/ Q x y)
   ==> (!x y. P x y) \/ (!x y. Q x y)`;;
```

First-order automation

HOL Light has a simple first-order prover `MESON` based on model elimination, which can dispose of much purely first-order reasoning, e.g.

```
MESON[]
  `(!x y z. P x y /\ P y z ==> P x z) /\
   (!x y z. Q x y /\ Q y z ==> Q x z) /\
   (!x y. P x y ==> P y x) /\
   (!x y. P x y \/ Q x y)
   ==> (!x y. P x y) \/ (!x y. Q x y)`;;
```

Cezary Kaliszyk and Josef Urban have created a much more powerful framework for first-order automation including many off-the-shelf first order provers and a framework for machine learning, which you can even use over a Web interface:

`http://cl-informatik.uibk.ac.at/software/hh/`

Basic set automation

HOL Light has a basic automated prover for facts of set theory: `SET_RULE`.

Basic set automation

HOL Light has a basic automated prover for facts of set theory: `SET_RULE`.

The code is basically trivial: rewrite away all the set operations and use first-order automation. Nevertheless it is extremely useful:

Basic set automation

HOL Light has a basic automated prover for facts of set theory: `SET_RULE`.

The code is basically trivial: rewrite away all the set operations and use first-order automation. Nevertheless it is extremely useful:

```
SET_RULE `t SUBSET s ==> t = s INTER t`;;

SET_RULE `~(s SUBSET {b}) <=> ?a. ~(a = b) /\ a IN s`;;

SET_RULE `(!x y. f x = f y ==> x = y) ==> (!x s. f x IN IMAGE f s <=> x IN s)`;
```

Basic set automation

HOL Light has a basic automated prover for facts of set theory: `SET_RULE`.
The code is basically trivial: rewrite away all the set operations and use first-order automation. Nevertheless it is extremely useful:

```
SET_RULE `t SUBSET s ==> t = s INTER t`;;

SET_RULE `~(s SUBSET {b}) <=> ?a. ~(a = b) /\ a IN s`;;

SET_RULE `(!x y. f x = f y ==> x = y) ==> (!x s. f x IN IMAGE f s <=> x IN s)`;
```

This is used frequently to generate such handy obvious facts that would otherwise be distracting in the middle of a real proof.

Algebra via Gröbner bases

HOL Light includes a Gröbner basis procedure which is at the core of several convenient algebraic rules like INT_RING, REAL_FIELD, COMPLEX_FIELD:

```
# REAL_FIELD `!x. &0 < x ==> &1 / x - &1 / (x + &1) = &1 / (x * (x + &1))`;;
val it : thm = |- !x. &0 < x ==> &1 / x - &1 / (x + &1) = &1 / (x * (x + &1))
```

Algebra via Gröbner bases

HOL Light includes a Gröbner basis procedure which is at the core of several convenient algebraic rules like `INT_RING`, `REAL_FIELD`, `COMPLEX_FIELD`:

```
# REAL_FIELD `!x. &0 < x ==> &1 / x - &1 / (x + &1) = &1 / (x * (x + &1))`;;
val it : thm = |- !x. &0 < x ==> &1 / x - &1 / (x + &1) = &1 / (x * (x + &1))
```

Here is "Vieta's substitution" for cubic equations, completely automatically:

```
REAL_RING
 `p = (&3 * a1 - a2 pow 2) / &3 /\
  q = (&9 * a1 * a2 - &27 * a0 - &2 * a2 pow 3) / &27 /\
  x = z + a2 / &3 /\
  x * w = w pow 2 - p / &3
  ==> (z pow 3 + a2 * z pow 2 + a1 * z + a0 = &0 <=>
       if p = &0 then x pow 3 = q
       else (w pow 3) pow 2 - q * (w pow 3) - p pow 3 / &27 = &0)`;;
```

Linear arithmetic

Basic facts of linear arithmetic are painful to prove by hand, but HOL Light has quite effective decision procedures for small cases.

Linear arithmetic

Basic facts of linear arithmetic are painful to prove by hand, but HOL Light has quite effective decision procedures for small cases. There is also a highly efficient implementation of linear programming due to Alexey Solovyev that is used extensively in Flyspeck.

Linear arithmetic

Basic facts of linear arithmetic are painful to prove by hand, but HOL Light has quite effective decision procedures for small cases. There is also a highly efficient implementation of linear programming due to Alexey Solovyev that is used extensively in Flyspeck.

```
# REAL_ARITH `!x y:real. x < y ==> x < (x + y) / &2 /\ (x + y) / &2 < y`;;
val it : thm = |- !x y. x < y ==> x < (x + y) / &2 /\ (x + y) / &2 < y

# REAL_ARITH `!x y:real. (abs(x) - abs(y)) <= abs(x - y)`;;
val it : thm = |- !x y. abs x - abs y <= abs (x - y)
```

Linear arithmetic

Basic facts of linear arithmetic are painful to prove by hand, but HOL Light has quite effective decision procedures for small cases. There is also a highly efficient implementation of linear programming due to Alexey Solovyev that is used extensively in Flyspeck.

```
# REAL_ARITH `!x y:real. x < y ==> x < (x + y) / &2 /\ (x + y) / &2 < y`;;
val it : thm = |- !x y. x < y ==> x < (x + y) / &2 /\ (x + y) / &2 < y

# REAL_ARITH `!x y:real. (abs(x) - abs(y)) <= abs(x - y)`;;
val it : thm = |- !x y. abs x - abs y <= abs (x - y)
```

These can also handle non-linear terms and division by constants in easy cases, e.g.

```
REAL_ARITH `(&1 + x) * (&1 - x) * (&1 + x pow 2) < &1 ==> &0 < x pow 4`;;

ARITH_RULE `x < 2 EXP 30 ==> (429496730 * x) DIV (2 EXP 32) = x DIV 10`;;
```

Linear arithmetic

Basic facts of linear arithmetic are painful to prove by hand, but HOL Light has quite effective decision procedures for small cases. There is also a highly efficient implementation of linear programming due to Alexey Solovyev that is used extensively in Flyspeck.

```
# REAL_ARITH `!x y:real. x < y ==> x < (x + y) / &2 /\ (x + y) / &2 < y`;;
val it : thm = |- !x y. x < y ==> x < (x + y) / &2 /\ (x + y) / &2 < y

# REAL_ARITH `!x y:real. (abs(x) - abs(y)) <= abs(x - y)`;;
val it : thm = |- !x y. abs x - abs y <= abs (x - y)
```

These can also handle non-linear terms and division by constants in easy cases, e.g.

```
REAL_ARITH `(&1 + x) * (&1 - x) * (&1 + x pow 2) < &1 ==> &0 < x pow 4`;;

ARITH_RULE `x < 2 EXP 30 ==> (429496730 * x) DIV (2 EXP 32) = x DIV 10`;;
```

However in general these are limited to linear problems and only (implicitly or explicitly) universal quantified formulas.

Quantifier elimination for linear arithmetic

`Examples/cooper.ml` has Cooper's algorithm for integer quantifier elimination as a derived rule, which can handle arbitrary quantifier structure:

```
# COOPER_RULE `!n. n >= 8 ==> ?a b. n = 3 * a + 5 * b`;;
val it : thm = |- !n. n >= 8 ==> (?a b. n = 3 * a + 5 * b)
```

Quantifier elimination for linear arithmetic

`Examples/cooper.ml` has Cooper's algorithm for integer quantifier elimination as a derived rule, which can handle arbitrary quantifier structure:

```
# COOPER_RULE `!n. n >= 8 ==> ?a b. n = 3 * a + 5 * b`;;
val it : thm = |- !n. n >= 8 ==> (?a b. n = 3 * a + 5 * b)
```

Here's an example where we can prove 'covering congruence' results more or less automatically:

```
let COVERING_CONGRUENCES_1 = prove
 (`!n. (n == 0) (mod 2) \/
       (n == 0) (mod 3) \/
       (n == 1) (mod 4) \/
       (n == 3) (mod 8) \/
       (n == 7) (mod 12) \/
       (n == 23) (mod 24)`,
  GEN_TAC THEN REWRITE_TAC[num_congruent; int_congruent] THEN
  SPEC_TAC(`&n:int`,`x:int`) THEN CONV_TAC COOPER_CONV);;
```

Quantifier elimination for real arithmetic

`Rqe` contains a derived quantifier elimination procedure for real arithmetic written by Sean McLaughlin. It is quite powerful in principle:

```
REAL_QELIM_CONV
 `!a b c. (?x. a * x pow 2 + b * x + c = &0) <=>
          a = &0 /\ (~(b = &0) \/ c = &0) \/
          ~(a = &0) /\ b pow 2 >= &4 * a * c`;;
```

Quantifier elimination for real arithmetic

Rqe contains a derived quantifier elimination procedure for real arithmetic written by Sean McLaughlin. It is quite powerful in principle:

```
REAL_QELIM_CONV
 `!a b c. (?x. a * x pow 2 + b * x + c = &0) <=>
          a = &0 /\ (~(b = &0) \/ c = &0) \/
          ~(a = &0) /\ b pow 2 >= &4 * a * c`;;
```

This seems to be one of the cases where insisting on full LCF-style proof generation really slows things down, so this can be quite time-consuming on large problems.

Nonlinear arithmetic using sum-of-squares

For purely *universal* nonlinear problems there is a procedure based on sums of squares (building on the work of Pablo Parrilo) which is often much more efficient.

Nonlinear arithmetic using sum-of-squares

For purely *universal* nonlinear problems there is a procedure based on sums of squares (building on the work of Pablo Parrilo) which is often much more efficient.

It relies on an external semidefinite programming engine like CSDP, but generates an algebraic certificate that can be verified very efficiently in HOL Light.

Nonlinear arithmetic using sum-of-squares

For purely *universal* nonlinear problems there is a procedure based on sums of squares (building on the work of Pablo Parrilo) which is often much more efficient.

It relies on an external semidefinite programming engine like CSDP, but generates an algebraic certificate that can be verified very efficiently in HOL Light.

```
# SOS_RULE `1 <= x /\ 1 <= y ==> 1 <= x * y`;;
val it : thm = |- 1 <= x /\ 1 <= y ==> 1 <= x * y
```

Nonlinear arithmetic using sum-of-squares

For purely *universal* nonlinear problems there is a procedure based on sums of squares (building on the work of Pablo Parrilo) which is often much more efficient.
It relies on an external semidefinite programming engine like CSDP, but generates an algebraic certificate that can be verified very efficiently in HOL Light.

```
# SOS_RULE `1 <= x /\ 1 <= y ==> 1 <= x * y`;;
val it : thm = |- 1 <= x /\ 1 <= y ==> 1 <= x * y
```

Under the surface the algebraic certificate involves rearranging expressions into sums of squares.

More SOS examples

There is also a conversion that will just explicitly rewrite expressions as sums of squares:

```
# SOS_CONV
   `&2 * x pow 4 + &2 * x pow 3 * y - x pow 2 * y pow 2 + &5 * y pow 4`;;
val it : thm =
  |- &2 * x pow 4 + &2 * x pow 3 * y - x pow 2 * y pow 2 + &5 * y pow 4 =
     &1 / &2 * (&2 * x pow 2 + x * y + -- &1 * y pow 2) pow 2 +
     &1 / &2 * (x * y + y pow 2) pow 2 +
     &4 * y pow 2 pow 2
```

More SOS examples

There is also a conversion that will just explicitly rewrite expressions as sums of squares:

```
# SOS_CONV
   `&2 * x pow 4 + &2 * x pow 3 * y - x pow 2 * y pow 2 + &5 * y pow 4`;;
val it : thm =
  |- &2 * x pow 4 + &2 * x pow 3 * y - x pow 2 * y pow 2 + &5 * y pow 4 =
     &1 / &2 * (&2 * x pow 2 + x * y + -- &1 * y pow 2) pow 2 +
     &1 / &2 * (x * y + y pow 2) pow 2 +
     &4 * y pow 2 pow 2
```

SOS is quite good at the kinds of inequalities you find in math olympiad problems:

```
REAL_SOS
 `!a b c:real.
        a >= &0 /\ b >= &0 /\ c >= &0
        ==> &3 / &2 * (b + c) * (a + c) * (a + b) <=
            a * (a + c) * (a + b) +
            b * (b + c) * (a + b) +
            c * (b + c) * (a + c)`;;
```

Nonlinear inequality reasoning with formal interval arithmetic

As part of the Flyspeck project Alexey Solovyev developed a highly efficient formal implementation of interval arithmetic (`Formal_ineqs`),

```
verify_ineq default_params 5
   `-- &10 <= x0 /\ x0 <= &40 /\ &40 <= x1 /\ x1 <= &100 /\
    -- &70 <= x2 /\ x2 <= -- &40 /\ -- &70 <= x3 /\ x3 <= &40 /\
       &10 <= x4 /\ x4 <= &20 /\ -- &10 <= x5 /\ x5 <= &20 /\
    -- &30 <= x6 /\ x6 <= &110 /\ -- &110 <= x7 /\ x7 <= -- &30
   ==> -- &1 * x0 * x5 pow 3 + &3 * x0 * x5 * x6 pow 2 - x2 * x6 pow 3 +
       &3 * x2 * x6 * x5 pow 2 - x1 * x4 pow 3 + &3 * x1 * x4 * x7 pow 2 -
       x3 * x7 pow 3 + &3 * x3 * x7 * x4 pow 2 - &9563453 / &10000000
       < &232480000`;;
```

Nonlinear inequality reasoning with formal interval arithmetic

As part of the Flyspeck project Alexey Solovyev developed a highly efficient formal implementation of interval arithmetic (`Formal_ineqs`),

```
verify_ineq default_params 5
   `-- &10 <= x0 /\ x0 <= &40 /\ &40 <= x1 /\ x1 <= &100 /\
    -- &70 <= x2 /\ x2 <= -- &40 /\ -- &70 <= x3 /\ x3 <= &40 /\
       &10 <= x4 /\ x4 <= &20 /\ -- &10 <= x5 /\ x5 <= &20 /\
    -- &30 <= x6 /\ x6 <= &110 /\ -- &110 <= x7 /\ x7 <= -- &30
   ==> -- &1 * x0 * x5 pow 3 + &3 * x0 * x5 * x6 pow 2 - x2 * x6 pow 3 +
       &3 * x2 * x6 * x5 pow 2 - x1 * x4 pow 3 + &3 * x1 * x4 * x7 pow 2 -
       x3 * x7 pow 3 + &3 * x3 * x7 * x4 pow 2 - &9563453 / &10000000
       < &232480000`;;
```

Besides being amazingly efficient, it can also handle several transcendental functions, e.g.

```
verify_ineq default_params 5
  `&0 <= x /\ x <= &1 ==> atn x - x / (&1 + #0.28 * x * x) < #0.005`;;
```

Divisibility properties

HOL Light has a convenient rule for proving a class of basic disibility properties over natural numbers

```
NUMBER_RULE
 `~(gcd(a,b) = 0) /\ a = a' * gcd(a,b) /\ b = b' * gcd(a,b)
  ==> coprime(a',b')`;;
```

Divisibility properties

HOL Light has a convenient rule for proving a class of basic disibility properties over natural numbers

```
NUMBER_RULE
 `~(gcd(a,b) = 0) /\ a = a' * gcd(a,b) /\ b = b' * gcd(a,b)
  ==> coprime(a',b')`;;
```

or integers

```
INTEGER_RULE `!x y. coprime(x * y,x pow 2 + y pow 2) <=> coprime(x,y)`;;

INTEGER_RULE `coprime(a,b) ==> ?x. (x == u) (mod a) /\ (x == v) (mod b)`;;
```

Divisibility properties

HOL Light has a convenient rule for proving a class of basic disibility properties over natural numbers

```
NUMBER_RULE
 `~(gcd(a,b) = 0) /\ a = a' * gcd(a,b) /\ b = b' * gcd(a,b)
  ==> coprime(a',b')`;;
```

or integers

```
INTEGER_RULE `!x y. coprime(x * y,x pow 2 + y pow 2) <=> coprime(x,y)`;;

INTEGER_RULE `coprime(a,b) ==> ?x. (x == u) (mod a) /\ (x == v) (mod b)`;;
```

Internally this is using Gröbner bases once again (see Harrison "Automating Elementary Number-Theoretic Proofs using Gröbner bases").

Tactic proofs

Goal-directed proofs

Another idea introduced by Milner in LCF was the use of *goal-directed* or *backward* proof.

Goal-directed proofs

Another idea introduced by Milner in LCF was the use of *goal-directed* or *backward* proof.

- Start with the goal to be proved and apply 'tactics' to break the goal into simpler subgoals, which eventually get solved.

Goal-directed proofs

Another idea introduced by Milner in LCF was the use of *goal-directed* or *backward* proof.

- Start with the goal to be proved and apply 'tactics' to break the goal into simpler subgoals, which eventually get solved.
- Internally, HOL Light remembers the corresponding proof and applies the forward rules once the proof is complete.

Goal-directed proofs

Another idea introduced by Milner in LCF was the use of *goal-directed* or *backward* proof.

- Start with the goal to be proved and apply 'tactics' to break the goal into simpler subgoals, which eventually get solved.
- Internally, HOL Light remembers the corresponding proof and applies the forward rules once the proof is complete.

Even with the use of powerful forward rules, most people find this goal-directed style more convenient. It is the usual way of proving results in HOL Light.

Setting up goals

HOL Light has a simple way (going back to Cambridge LCF) of setting up a "current goal" and applying tactics.

Setting up goals

HOL Light has a simple way (going back to Cambridge LCF) of setting up a "current goal" and applying tactics.

A new goal can be established using g:

```
g `x >= x - 3 /\ (f(x + 1) + 3 < f(y + 1) + 3 ==> ~(x = y))`;;
```

Setting up goals

HOL Light has a simple way (going back to Cambridge LCF) of setting up a "current goal" and applying tactics.

A new goal can be established using g:

```
g `x >= x - 3 /\ (f(x + 1) + 3 < f(y + 1) + 3 ==> ~(x = y))`;;
```

Apply tactics using e ("expand"), e.g. `CONJ_TAC` that breaks a conjunctive goal into two conjuncts:

```
# e CONJ_TAC;;
val it : goalstack = 2 subgoals (2 total)

`f (x + 1) + 3 < f (y + 1) + 3 ==> ~(x = y)`

`x >= x - 3`
```

Solving subgoals

Solving subgoals

We can solve the first subgoal with ARITH_TAC (a tactic variant of ARITH_RULE)

```
#  e ARITH_TAC;;
val it : goalstack = 1 subgoal (1 total)

`f (x + 1) + 3 < f (y + 1) + 3 ==> ~(x = y)`
```

Solving subgoals

We can solve the first subgoal with ARITH_TAC (a tactic variant of ARITH_RULE)

```
#  e ARITH_TAC;;
val it : goalstack = 1 subgoal (1 total)

`f (x + 1) + 3 < f (y + 1) + 3 ==> ~(x = y)`
```

and the other with first-order logic noting the fact that $<$ is irreflexive

```
# e(MESON_TAC[LT_REFL]);;
0..0..solved at 2
val it : goalstack = No subgoals
```

Solving subgoals

We can solve the first subgoal with `ARITH_TAC` (a tactic variant of `ARITH_RULE`)

```
#  e ARITH_TAC;;
val it : goalstack = 1 subgoal (1 total)

`f (x + 1) + 3 < f (y + 1) + 3 ==> ~(x = y)`
```

and the other with first-order logic noting the fact that $<$ is irreflexive

```
# e(MESON_TAC[LT_REFL]);;
0..0..solved at 2
val it : goalstack = No subgoals
```

We can get at the final theorem now all goals are solved with `top_thm()`

```
# top_thm();;
val it : thm = |- x >= x - 3 /\ (f (x + 1) + 3 < f (y + 1) + 3 ==> ~(x = y))
```

Converting rules to tactics

Many forward inference rules have tactic variants, and those that don't can often be converted by `CONV_TAC`, which takes either

Converting rules to tactics

Many forward inference rules have tactic variants, and those that don't can often be converted by CONV_TAC, which takes either

- A rule that proves a proposition like CONV_RULE

Converting rules to tactics

Many forward inference rules have tactic variants, and those that don't can often be converted by `CONV_TAC`, which takes either

- A rule that proves a proposition like `CONV_RULE`
- A rule (called a *conversion* that proves a term equal to another one)

Converting rules to tactics

Many forward inference rules have tactic variants, and those that don't can often be converted by `CONV_TAC`, which takes either

- A rule that proves a proposition like `CONV_RULE`
- A rule (called a *conversion* that proves a term equal to another one)

and applies it in a tactic framework, e.g. `CONV_TAC REAL_ARITH`.

The duality between rules and tactics

Most of the (primitive or derived) logical inference that work forward on theorems like `CONJ`:

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

The duality between rules and tactics

Most of the (primitive or derived) logical inference that work forward on theorems like CONJ:

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

have natural tactic variants (here CONJ_TAC) that apply the rule 'backwards'.

Some useful tactics

Some useful tactics

- `REWRITE_TAC` and `ASM_REWRITE_TAC` — rewrite the goal with a list of theorems (including the assumptions).

Some useful tactics

- REWRITE_TAC and ASM_REWRITE_TAC — rewrite the goal with a list of theorems (including the assumptions).
- SIMP_TAC and ASM_SIMP_TAC — more powerful versions of rewriting using context

Some useful tactics

- `REWRITE_TAC` and `ASM_REWRITE_TAC` — rewrite the goal with a list of theorems (including the assumptions).
- `SIMP_TAC` and `ASM_SIMP_TAC` — more powerful versions of rewriting using context
- `MATCH_MP_TAC` — use a theorem of the form $\vdash p \Rightarrow q$ with matching to reduce goal q' to p'

Some useful tactics

- REWRITE_TAC and ASM_REWRITE_TAC — rewrite the goal with a list of theorems (including the assumptions).
- SIMP_TAC and ASM_SIMP_TAC — more powerful versions of rewriting using context
- MATCH_MP_TAC — use a theorem of the form $\vdash p \Rightarrow q$ with matching to reduce goal q' to p'
- INDUCT_TAC — apply induction on natural numbers

Some useful tactics

- REWRITE_TAC and ASM_REWRITE_TAC — rewrite the goal with a list of theorems (including the assumptions).
- SIMP_TAC and ASM_SIMP_TAC — more powerful versions of rewriting using context
- MATCH_MP_TAC — use a theorem of the form $\vdash p \Rightarrow q$ with matching to reduce goal q' to p'
- INDUCT_TAC — apply induction on natural numbers
- STRIP_TAC — break down a goal moving hypotheses into assumption list etc.

Some useful tactics

- REWRITE_TAC and ASM_REWRITE_TAC — rewrite the goal with a list of theorems (including the assumptions).
- SIMP_TAC and ASM_SIMP_TAC — more powerful versions of rewriting using context
- MATCH_MP_TAC — use a theorem of the form $\vdash p \Rightarrow q$ with matching to reduce goal q' to p'
- INDUCT_TAC — apply induction on natural numbers
- STRIP_TAC — break down a goal moving hypotheses into assumption list etc.
- ASSUME_TAC and MP_TAC — introduce an existing theorem as a hypothesis

There are also 'tacticals' for combining tactics in various ways, e.g. THEN to apply them one after the other, REPEAT to apply them repeatedly.

A simple example (1)

Let's prove the formula for the sum of the first n natural numbers:

```
# g `!n. nsum(1..n) (\i. i) = (n * (n + 1)) DIV 2`;;
val it : goalstack = 1 subgoal (1 total)

`!n. nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2`
```

A simple example (1)

Let's prove the formula for the sum of the first n natural numbers:

```
# g `!n. nsum(1..n) (\i. i) = (n * (n + 1)) DIV 2`;;
val it : goalstack = 1 subgoal (1 total)

`!n. nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2`
```

We apply induction and rewrite both goals with the recursive definition of sums:

```
# e(INDUCT_TAC THEN REWRITE_TAC[NSUM_CLAUSES_NUMSEG]);;
val it : goalstack = 2 subgoals (2 total)

  0 [`nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2`]

`(if 1 <= SUC n then nsum (1..n) (\i. i) + SUC n else nsum (1..n) (\i. i)) =
 (SUC n * (SUC n + 1)) DIV 2`

`(if 1 = 0 then 0 else 0) = (0 * (0 + 1)) DIV 2`
```

A simple example (2)

The first goal is trivial

```
# e ARITH_TAC;;
val it : goalstack = 1 subgoal (1 total)

  0 [`nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2`]

`(if 1 <= SUC n then nsum (1..n) (\i. i) + SUC n else nsum (1..n) (\i. i)) =
 (SUC n * (SUC n + 1)) DIV 2`
```

A simple example (2)

The first goal is trivial

```
# e ARITH_TAC;;
val it : goalstack = 1 subgoal (1 total)

  0 [`nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2`]

`(if 1 <= SUC n then nsum (1..n) (\i. i) + SUC n else nsum (1..n) (\i. i)) =
 (SUC n * (SUC n + 1)) DIV 2`
```

The other one can be solved by `ASM_ARITH_TAC`, or we can first rewrite with the assumptions via `ASM_REWRITE_TAC` then use `ARITH_TAC` again:

```
# e(ASM_REWRITE_TAC[] THEN ARITH_TAC);;

val it : goalstack = No subgoals
```

A simple example (2)

The first goal is trivial

```
# e ARITH_TAC;;
val it : goalstack = 1 subgoal (1 total)

  0 [`nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2`]

`(if 1 <= SUC n then nsum (1..n) (\i. i) + SUC n else nsum (1..n) (\i. i)) =
 (SUC n * (SUC n + 1)) DIV 2`
```

The other one can be solved by `ASM_ARITH_TAC`, or we can first rewrite with the assumptions via `ASM_REWRITE_TAC` then use `ARITH_TAC` again:

```
# e(ASM_REWRITE_TAC[] THEN ARITH_TAC);;

val it : goalstack = No subgoals
```

and so

```
# top_thm();;
val it : thm = |- !n. nsum (1..n) (\i. i) = (n * (n + 1)) DIV 2
```

Packaging tactic proofs

Even if they are developed interactively via 'g' and 'e' steps, it's common to package up the tactics into blocks using a `prove` function.

```
let OUR_LEMMA = prove
 (`!n. nsum(1..n) (\i. i) = (n * (n + 1)) DIV 2`,
  INDUCT_TAC THEN ASM_REWRITE_TAC[NSUM_CLAUSES_NUMSEG] THEN ARITH_TAC);;
```

Packaging tactic proofs

Even if they are developed interactively via ‘g’ and ‘e’ steps, it’s common to package up the tactics into blocks using a `prove` function.

```
let OUR_LEMMA = prove
 (`!n. nsum(1..n) (\i. i) = (n * (n + 1)) DIV 2`,
  INDUCT_TAC THEN ASM_REWRITE_TAC[NSUM_CLAUSES_NUMSEG] THEN ARITH_TAC);;
```

I tend to construct the proof in this format in the editor as I work and just paste it into HOL interactively. Mark Adams has a tool called *Tactician* for converting between the forms:

`http://www.proof-technologies.com/tactician/`

Packaging tactic proofs

Even if they are developed interactively via 'g' and 'e' steps, it's common to package up the tactics into blocks using a `prove` function.

```
let OUR_LEMMA = prove
 (`!n. nsum(1..n) (\i. i) = (n * (n + 1)) DIV 2`,
  INDUCT_TAC THEN ASM_REWRITE_TAC[NSUM_CLAUSES_NUMSEG] THEN ARITH_TAC);;
```

I tend to construct the proof in this format in the editor as I work and just paste it into HOL interactively. Mark Adams has a tool called *Tactician* for converting between the forms:

`http://www.proof-technologies.com/tactician/`

For a video of me proving a slightly larger theorem interactively in a competition, see

`http://www.math.kobe-u.ac.jp/icms2006/icms2006-video/video/v103.html`

A tour of the library

Some of the basic library files

HOL Light has quite a few library files developing some branches of mathematics in more detail, e.g.

- `Library/prime.ml` and `Library/pocklington.ml` — divisibility properties, prime numbers, certifying the primality of particular numbers

Some of the basic library files

HOL Light has quite a few library files developing some branches of mathematics in more detail, e.g.

- `Library/prime.ml` and `Library/pocklington.ml` — divisibility properties, prime numbers, certifying the primality of particular numbers
- `Library/card.ml` — Notions of cardinal arithmetic, just using injections and surjections to compare sets.

Some of the basic library files

HOL Light has quite a few library files developing some branches of mathematics in more detail, e.g.

- `Library/prime.ml` and `Library/pocklington.ml` — divisibility properties, prime numbers, certifying the primality of particular numbers
- `Library/card.ml` — Notions of cardinal arithmetic, just using injections and surjections to compare sets.
- `Library/wo.ml` — Common Axiom of Choice equivalents like the wellordering principle and Zorn's lemma

Some of the basic library files

HOL Light has quite a few library files developing some branches of mathematics in more detail, e.g.

- `Library/prime.ml` and `Library/pocklington.ml` — divisibility properties, prime numbers, certifying the primality of particular numbers
- `Library/card.ml` — Notions of cardinal arithmetic, just using injections and surjections to compare sets.
- `Library/wo.ml` — Common Axiom of Choice equivalents like the wellordering principle and Zorn's lemma
- `Library/rstc.ml` — Reflexive, symmetric and transitive closures of binary relations.

More substantial library components

The following are a few of the extended developments with a directory of their own:

More substantial library components

The following are a few of the extended developments with a directory of their own:

- `Boyer_Moore` — Boyer-Moore style automation (Petros Papapanagiotou)

More substantial library components

The following are a few of the extended developments with a directory of their own:

- `Boyer_Moore` — Boyer-Moore style automation (Petros Papapanagiotou)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)

More substantial library components

The following are a few of the extended developments with a directory of their own:

- `Boyer_Moore` — Boyer-Moore style automation (Petros Papapanagiotou)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)

More substantial library components

The following are a few of the extended developments with a directory of their own:

- `Boyer_Moore` — Boyer-Moore style automation (Petros Papapanagiotou)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)
- `Permutation` — theory of list permutations (Marco Maggesi)

More substantial library components

The following are a few of the extended developments with a directory of their own:

- `Boyer_Moore` — Boyer-Moore style automation (Petros Papapanagiotou)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)
- `Permutation` — theory of list permutations (Marco Maggesi)
- `Proofrecording` — system for recording and exporting proofs (Steven Obua, Chantal Keller)

More substantial library components

The following are a few of the extended developments with a directory of their own:

- `Boyer_Moore` — Boyer-Moore style automation (Petros Papapanagiotou)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)
- `Permutation` — theory of list permutations (Marco Maggesi)
- `Proofrecording` — system for recording and exporting proofs (Steven Obua, Chantal Keller)
- `QBF` — proving quantified Boolean formulas (Ondřej Kunčar)

More substantial library components

The following are a few of the extended developments with a directory of their own:

- `Boyer_Moore` — Boyer-Moore style automation (Petros Papapanagiotou)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)
- `Permutation` — theory of list permutations (Marco Maggesi)
- `Proofrecording` — system for recording and exporting proofs (Steven Obua, Chantal Keller)
- `QBF` — proving quantified Boolean formulas (Ondřej Kunčar)
- `Quaternions` — theory of quaternions (Marco Maggesi)

More substantial library components

The following are a few of the extended developments with a directory of their own:

- `Boyer_Moore` — Boyer-Moore style automation (Petros Papapanagiotou)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)
- `Permutation` — theory of list permutations (Marco Maggesi)
- `Proofrecording` — system for recording and exporting proofs (Steven Obua, Chantal Keller)
- `QBF` — proving quantified Boolean formulas (Ondřej Kunčar)
- `Quaternions` — theory of quaternions (Marco Maggesi)
- `RichterHilbertAxiomGeometry` — geometry proofs in a readable format (Bill Richter)

More substantial library components

The following are a few of the extended developments with a directory of their own:

- `Boyer_Moore` — Boyer-Moore style automation (Petros Papapanagiotou)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)
- `Permutation` — theory of list permutations (Marco Maggesi)
- `Proofrecording` — system for recording and exporting proofs (Steven Obua, Chantal Keller)
- `QBF` — proving quantified Boolean formulas (Ondřej Kunčar)
- `Quaternions` — theory of quaternions (Marco Maggesi)
- `RichterHilbertAxiomGeometry` — geometry proofs in a readable format (Bill Richter)
- `Unity` — Chandy-Misra Unity theory (Flemming Andersen)

Some “great 100 theorems”

`http://www.cs.ru.nl/~freek/100/`

Some “great 100 theorems”

`http://www.cs.ru.nl/~freek/100/`

HOL Light currently has 86 of them; those that are not already buried in other library files are in the subdirectory 100, e.g.

Some "great 100 theorems"

`http://www.cs.ru.nl/~freek/100/`

HOL Light currently has 86 of them; those that are not already buried in other library files are in the subdirectory 100, e.g.

- `100/cayley_hamilton.ml` — The Cayley-Hamilton theorem

Some "great 100 theorems"

`http://www.cs.ru.nl/~freek/100/`

HOL Light currently has 86 of them; those that are not already buried in other library files are in the subdirectory 100, e.g.

- `100/cayley_hamilton.ml` — The Cayley-Hamilton theorem
- `100/constructible.ml` — Impossibility of angle trisection or cube construction using geometric constructions

Some "great 100 theorems"

`http://www.cs.ru.nl/~freek/100/`

HOL Light currently has 86 of them; those that are not already buried in other library files are in the subdirectory 100, e.g.

- `100/cayley_hamilton.ml` — The Cayley-Hamilton theorem
- `100/constructible.ml` — Impossibility of angle trisection or cube construction using geometric constructions
- `100/dirichlet.ml` — Dirichlet's theorem on primes in arithmetic progression

Some "great 100 theorems"

`http://www.cs.ru.nl/~freek/100/`

HOL Light currently has 86 of them; those that are not already buried in other library files are in the subdirectory 100, e.g.

- `100/cayley_hamilton.ml` — The Cayley-Hamilton theorem
- `100/constructible.ml` — Impossibility of angle trisection or cube construction using geometric constructions
- `100/dirichlet.ml` — Dirichlet's theorem on primes in arithmetic progression
- `100/e_is_transcendental.ml` — Proof that e is transcendental (Jesse Bingham)

Some "great 100 theorems"

`http://www.cs.ru.nl/~freek/100/`

HOL Light currently has 86 of them; those that are not already buried in other library files are in the subdirectory 100, e.g.

- `100/cayley_hamilton.ml` — The Cayley-Hamilton theorem
- `100/constructible.ml` — Impossibility of angle trisection or cube construction using geometric constructions
- `100/dirichlet.ml` — Dirichlet's theorem on primes in arithmetic progression
- `100/e_is_transcendental.ml` — Proof that e is transcendental (Jesse Bingham)
- `100/fourier.ml` — Basic results about Fourier series

Some "great 100 theorems"

`http://www.cs.ru.nl/~freek/100/`

HOL Light currently has 86 of them; those that are not already buried in other library files are in the subdirectory 100, e.g.

- `100/cayley_hamilton.ml` — The Cayley-Hamilton theorem
- `100/constructible.ml` — Impossibility of angle trisection or cube construction using geometric constructions
- `100/dirichlet.ml` — Dirichlet's theorem on primes in arithmetic progression
- `100/e_is_transcendental.ml` — Proof that e is transcendental (Jesse Bingham)
- `100/fourier.ml` — Basic results about Fourier series
- `100/minkowski.ml` — Minkowski's classic geometry of numbers theorem

Some "great 100 theorems"

`http://www.cs.ru.nl/~freek/100/`

HOL Light currently has 86 of them; those that are not already buried in other library files are in the subdirectory 100, e.g.

- `100/cayley_hamilton.ml` — The Cayley-Hamilton theorem
- `100/constructible.ml` — Impossibility of angle trisection or cube construction using geometric constructions
- `100/dirichlet.ml` — Dirichlet's theorem on primes in arithmetic progression
- `100/e_is_transcendental.ml` — Proof that e is transcendental (Jesse Bingham)
- `100/fourier.ml` — Basic results about Fourier series
- `100/minkowski.ml` — Minkowski's classic geometry of numbers theorem
- `100/pnt.ml` — The Prime Number Theorem

Some “great 100 theorems”

`http://www.cs.ru.nl/~freek/100/`

HOL Light currently has 86 of them; those that are not already buried in other library files are in the subdirectory 100, e.g.

- `100/cayley_hamilton.ml` — The Cayley-Hamilton theorem
- `100/constructible.ml` — Impossibility of angle trisection or cube construction using geometric constructions
- `100/dirichlet.ml` — Dirichlet's theorem on primes in arithmetic progression
- `100/e_is_transcendental.ml` — Proof that e is transcendental (Jesse Bingham)
- `100/fourier.ml` — Basic results about Fourier series
- `100/minkowski.ml` — Minkowski's classic geometry of numbers theorem
- `100/pnt.ml` — The Prime Number Theorem
- `100/polyhedron.ml` — Euler's polyhedron formula $V + F - E = 2$

The Multivariate library

Partly as a result of Flyspeck, HOL Light is particularly strong in the area of topology, analysis and geometry in Euclidean space $\mathbb{R}^n$.

The Multivariate library

Partly as a result of Flyspeck, HOL Light is particularly strong in the area of topology, analysis and geometry in Euclidean space $\mathbb{R}^n$.

File	Lines	Contents
misc.ml	756	Background stuff
metric .ml	2566	Metric spaces and general topology
vectors.ml	9789	Basic vectors, linear algebra
determinants.ml	3797	Determinant and trace
topology.ml	25105	Topology of euclidean space
convex.ml	15509	Convex sets and functions
paths.ml	19900	Paths, simple connectedness etc.
polytope.ml	8890	Faces, polytopes, polyhedra etc.
degree.ml	9066	Degree theory, retracts etc.
derivatives.ml	2885	Derivatives
clifford.ml	979	Geometric (Clifford) algebra
integration.ml	22362	Integration
measure.ml	20264	Lebesgue measure

Multivariate theories continued

From this foundation complex analysis is developed and used to derive convenient theorems for $\mathbb{R}$ as well as more topological results.

File	Lines	Contents
complexes.ml	2051	Complex numbers
canal.ml	3756	Complex analysis
transcendentals.ml	7584	Real & complex transcendentals
realanalysis.ml	16620	Some analytical stuff on $\mathbb{R}$
moretop.ml	7216	Further topological results
cauchy.ml	19771	Complex line integrals

Multivariate theories continued

From this foundation complex analysis is developed and used to derive convenient theorems for $\mathbb{R}$ as well as more topological results.

File	Lines	Contents
complexes.ml	2051	Complex numbers
canal.ml	3756	Complex analysis
transcendentals.ml	7584	Real & complex transcendentals
realanalysis.ml	16620	Some analytical stuff on $\mathbb{R}$
moretop.ml	7216	Further topological results
cauchy.ml	19771	Complex line integrals

Credits: JRH, Marco Maggesi, Valentina Bruno, Graziano Gentili, Gianni Ciolli, Lars Schewe, . . .

Multivariate theories continued

From this foundation complex analysis is developed and used to derive convenient theorems for $\mathbb{R}$ as well as more topological results.

File	Lines	Contents
complexes.ml	2051	Complex numbers
canal.ml	3756	Complex analysis
transcendentals.ml	7584	Real & complex transcendentals
realanalysis.ml	16620	Some analytical stuff on $\mathbb{R}$
moretop.ml	7216	Further topological results
cauchy.ml	19771	Complex line integrals

Credits: JRH, Marco Maggesi, Valentina Bruno, Graziano Gentili, Gianni Ciolli, Lars Schewe, ...

It would be desirable to generalize more of the material to general topological spaces, metric spaces, measure spaces etc.

Some examples from topology

The Brouwer fixed point theorem:

```
|- !f:real^N->real^N s.
      compact s /\ convex s /\ ~(s = {}) /\
      f continuous_on s /\ IMAGE f s SUBSET s
      ==> ?x. x IN s /\ f x = x
```

Some examples from topology

The Brouwer fixed point theorem:

```
|- !f:real^N->real^N s.
      compact s /\ convex s /\ ~(s = {}) /\
      f continuous_on s /\ IMAGE f s SUBSET s
      ==> ?x. x IN s /\ f x = x
```

The Borsuk homotopy extension theorem:

```
|- !f:real^M->real^N g s t u.
        closed_in (subtopology euclidean t) s /\
        (ANR s /\ ANR t \/ ANR u) /\
        f continuous_on t /\ IMAGE f t SUBSET u /\
        homotopic_with (\x. T) (s,u) f g
        ==> ?g'. homotopic_with (\x. T) (t,u) f g' /\
                 g' continuous_on t /\
                 IMAGE g' t SUBSET u /\
                 !x. x IN s ==> g'(x) = g(x)
```

Some examples from convexity

The Krein-Milman (Minkowski) theorem

```
|- !s:real^N->bool.
        convex s /\ compact s
        ==> s = convex hull {x | x extreme_point_of s}
```

Some examples from convexity

The Krein-Milman (Minkowski) theorem

```
|- !s:real^N->bool.
        convex s /\ compact s
        ==> s = convex hull {x | x extreme_point_of s}
```

Approximation of convex sets by polytopes w.r.t. Hausdorff distance:

```
|- !s:real^N->bool e.
        bounded s /\ convex s /\ &0 < e
        ==> ?p. polytope p /\ s SUBSET p /\ hausdist(p,s) < e
```

Some Lipschitz/derivative examples

Kirszbraun's theorem on extension of Lipschitz functions:

```
|- !f:real^M->real^N s B.
      &0 <= B /\
      (!x y. x IN s /\ y IN s ==> norm(f x - f y) <= B * norm(x - y))
      ==> (?g. (!x y. norm(g x - g y) <= B * norm(x - y)) /\
               (!x. x IN s ==> g x = f x))
```

Some Lipschitz/derivative examples

Kirszbraun's theorem on extension of Lipschitz functions:

```
|- !f:real^M->real^N s B.
      &0 <= B /\
      (!x y. x IN s /\ y IN s ==> norm(f x - f y) <= B * norm(x - y))
      ==> (?g. (!x y. norm(g x - g y) <= B * norm(x - y)) /\
               (!x. x IN s ==> g x = f x))
```

The Lebesgue differentiation theorem

```
|- !f:real^1->real^N s.
      is_interval s /\ f has_bounded_variation_on s
      ==> negligible {x | x IN s /\ ~(f differentiable at x)}
```

Some examples from measure theory

Steinhaus's theorem:

```
|- !s:real^N->bool.
        lebesgue_measurable s /\ ~negligible s
        ==> ?d. &0 < d /\ ball(vec 0,d) SUBSET {x - y | x IN s /\ y IN s}
```

Some examples from measure theory

Steinhaus's theorem:

```
|- !s:real^N->bool.
        lebesgue_measurable s /\ ~negligible s
        ==> ?d. &0 < d /\ ball(vec 0,d) SUBSET {x - y | x IN s /\ y IN s}
```

Luzin's theorem:

```
|- !f:real^M->real^N s e.
        measurable s /\ f measurable_on s /\ &0 < e
        ==> ?k. compact k /\ k SUBSET s /\ measure(s DIFF k) < e /\
                f continuous_on k
```

Some examples from complex analysis

The Little Picard theorem:

```
|- !f:complex->complex a b.
      f holomorphic_on (:complex) /\
      ~(a = b) /\ IMAGE f (:complex) INTER {a,b} = {}
      ==> ?c. f = \x. c
```

Some examples from complex analysis

The Little Picard theorem:

```
|- !f:complex->complex a b.
      f holomorphic_on (:complex) /\
      ~(a = b) /\ IMAGE f (:complex) INTER {a,b} = {}
      ==> ?c. f = \x. c
```

The Riemann mapping theorem:

```
|- !s:complex->bool.
        open s /\ simply_connected s <=>
        s = {} \/ s = (:complex) \/
        ?f g. f holomorphic_on s /\
              g holomorphic_on ball(Cx(&0),&1) /\
              (!z. z IN s ==> f z IN ball(Cx(&0),&1) /\ g(f z) = z) /\
              (!z. z IN ball(Cx(&0),&1) ==> g z IN s /\ f(g z) = z)
```

Thank you!