HOL Light from the foundations (part 2/3)

John Harrison

Amazon Web Services

22st Sep 2023 (09:00–10:30)

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 $\alpha \rightarrow \mathsf{bool}$.

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 $\alpha \to \mathsf{bool}$.
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 $\alpha \to \mathsf{bool}$.
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 more 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, METIS)
- Basic set theory
- Algebra via Gröbner bases
- Linear arithmetic
- ...

More automated derived rules

HOL Light does have quite a few more 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, METIS)
- 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)`;;
```

There is also an analogous `METIS` due to Joe Hurd, as well as an experimental “Hammer” (Cezary Kaliszyk and Josef Urban) using external provers together with machine learning:

`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

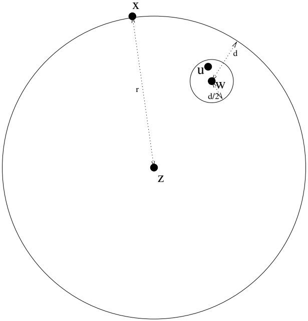
```
INTEGER_RULE `!x y. coprime(x * y,x pow 2 + y pow 2) <=> coprime(x,y)`;;

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").

Normed space procedure

We also have convenient 'linear decision procedure' for both normed spaces and metric spaces (latter from Marco Maggesi), analogous to the typical ones for integers, reals etc.

```
NORM_ARITH
 `abs(norm(w - z) - r) = d /\ norm(u - w) < d / &2 /\ norm(x - z) = r
   ==> d / &2 <= norm(x - u)`;;
```



See Solovay, Arthan and Harrison *Some new results on decidability for elementary algebra and geometry*

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`:

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

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)
- `Divstep` — bounds proof for optimized binary gcd (Dan Bernstein)

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)
- `Divstep` — bounds proof for optimized binary gcd (Dan Bernstein)
- `EC` — Common elliptic curves for cryptography

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)
- `Divstep` — bounds proof for optimized binary gcd (Dan Bernstein)
- `EC` — Common elliptic curves for cryptography
- `GL` — Gödel-Löb modal logic of provability (Cosimo Brogi)

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)
- `Divstep` — bounds proof for optimized binary gcd (Dan Bernstein)
- `EC` — Common elliptic curves for cryptography
- `GL` — Gödel-Löb modal logic of provability (Cosimo Brogi)
- `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)
- `Divstep` — bounds proof for optimized binary gcd (Dan Bernstein)
- `EC` — Common elliptic curves for cryptography
- `GL` — Gödel-Löb modal logic of provability (Cosimo Brogi)
- `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)
- `Divstep` — bounds proof for optimized binary gcd (Dan Bernstein)
- `EC` — Common elliptic curves for cryptography
- `GL` — Gödel-Löb modal logic of provability (Cosimo Brogi)
- `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)
- `Divstep` — bounds proof for optimized binary gcd (Dan Bernstein)
- `EC` — Common elliptic curves for cryptography
- `GL` — Gödel-Löb modal logic of provability (Cosimo Brogi)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)
- `Permutation` — theory of list permutations (Marco Maggesi)
- `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)
- `Divstep` — bounds proof for optimized binary gcd (Dan Bernstein)
- `EC` — Common elliptic curves for cryptography
- `GL` — Gödel-Löb modal logic of provability (Cosimo Brogi)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)
- `Permutation` — theory of list permutations (Marco Maggesi)
- `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)
- `Divstep` — bounds proof for optimized binary gcd (Dan Bernstein)
- `EC` — Common elliptic curves for cryptography
- `GL` — Gödel-Löb modal logic of provability (Cosimo Brogi)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)
- `Permutation` — theory of list permutations (Marco Maggesi)
- `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)
- `Divstep` — bounds proof for optimized binary gcd (Dan Bernstein)
- `EC` — Common elliptic curves for cryptography
- `GL` — Gödel-Löb modal logic of provability (Cosimo Brogi)
- `IEEE` — IEEE floating-point arithmetic (Charlie Jacobsen)
- `miz3`, `Mizarlight` — Frameworks for declarative proofs (Freek Wiedijk)
- `Permutation` — theory of list permutations (Marco Maggesi)
- `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 87 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 87 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 87 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 87 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 87 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 87 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 87 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/isoperimetric.ml` — The Isoperimetric Theorem

Some “great 100 theorems”

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

HOL Light currently has 87 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/isoperimetric.ml` — The Isoperimetric Theorem
- `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 87 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/isoperimetric.ml` — The Isoperimetric Theorem
- `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 87 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/isoperimetric.ml` — The Isoperimetric Theorem
- `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	2594	Background stuff
metric .ml	35321	Metric spaces and general topology
vectors.ml	10923	Basic vectors, linear algebra
determinants.ml	4956	Determinant and trace
topology.ml	36653	Topology of euclidean space
convex.ml	18279	Convex sets and functions
paths.ml	29932	Paths, simple connectedness etc.
polytope.ml	8940	Faces, polytopes, polyhedra etc.
degree.ml	9706	Degree theory, retracts etc.
derivatives.ml	5797	Derivatives
clifford.ml	979	Geometric (Clifford) algebra
integration.ml	26193	Integration
measure.ml	32007	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	2249	Complex numbers
canal.ml	4031	Complex analysis
transcendentals.ml	7590	Real & complex transcendentals
realanalysis.ml	17718	Some analytical stuff on $\mathbb{R}$
moretop.ml	7850	Further topological results
cauchy.ml	24103	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	2249	Complex numbers
canal.ml	4031	Complex analysis
transcendentals.ml	7590	Real & complex transcendentals
realanalysis.ml	17718	Some analytical stuff on $\mathbb{R}$
moretop.ml	7850	Further topological results
cauchy.ml	24103	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	2249	Complex numbers
canal.ml	4031	Complex analysis
transcendentals.ml	7590	Real & complex transcendentals
realanalysis.ml	17718	Some analytical stuff on $\mathbb{R}$
moretop.ml	7850	Further topological results
cauchy.ml	24103	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!