The Metis Theorem Prover

Joe Hurd

Galois, Inc.
joe@galois.com

Strategic CAD Labs, Intel
Thursday 17 April 2008

Talk Plan

A Familiar Beginning

Code (Reverse.hs)

```
import Test.QuickCheck(quickCheck)

rev :: [a] -> [a]
rev [] = []
rev (h:t) = rev t ++ [h]

prop :: [Int] -> Bool
prop l = rev (rev l) == l

quickCheck prop
```

Shell

```
$ ghc -package QuickCheck -o reverse Reverse.hs
$ ./reverse
OK, passed 100 tests.
```

From Bug-Finding to Assurance

- *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence"* [Dijkstra, The Humble Programmer]
- How can we do better?
- Formal verification:
  1. Model the Haskell program in a logic.
  2. Prove that it satisfies the property.
  3. Machine check the proof.
- Perhaps a library of verified functions could be built this way?

A Logic for Haskell Programs

Finding a suitable logic for Haskell programs is a whole other talk.

- For the sake of an example, will use Higher Order Logic of Computable Functions (a.k.a. HOLCF, a.k.a. domain theory).

Axioms

- $\mathrm{rev}\ [] = []$
- $\forall h, t.\ \mathrm{rev}\ (h : t) = \mathrm{rev}\ t +\!\!+ [h]$

Goal

$$\forall l.\ \mathrm{finite}\ l \implies \mathrm{rev}\ (\mathrm{rev}\ l) = l$$

Automation Hazard: Creative Step Required!

First need to generalize the goal to make it inductively provable:

Goal (Generalized goal)

$$\forall l, k.\ \text{finite}\ l\ \wedge\ \text{finite}\ k\ \Longrightarrow\ \text{rev}\ (\text{rev}\ l \mathbin{+\!\!+} k) = \text{rev}\ k \mathbin{+\!\!+} l$$

(Instantiate k to [] to recover the original goal.)

- Automatic generalization is hard.
- To have a reliable QuickCheck-like interface, the programmer would need to provide generalizations as hints.

List Induction Step

Now a standard induction over finite lists can be applied:

Goal (Base case)

$$\forall k.\ \text{finite } k \implies \text{rev}\ (\text{rev}\ []\ +\!\!+\ k) = \text{rev}\ k\ +\!\!+\ []$$

Goal (Step case)

$$\begin{array}{l} \forall t.\ \text{finite } t \implies \\ \quad (\forall k.\ \text{finite } k \implies \text{rev}\ (\text{rev}\ t\ +\!\!+\ k) = \text{rev}\ k\ +\!\!+\ t) \implies \\ \quad \forall h, k.\ \text{finite } k \implies \text{rev}\ (\text{rev}\ (h : t)\ +\!\!+\ k) = \text{rev}\ k\ +\!\!+\ (h : t) \end{array}$$

Include Relevant Facts

Some extra facts need to be included to prove the goals:

Axioms

- finite []
- $\forall h, t.\ \text{finite}\ (h : t) \iff \text{finite}\ t$
- $\forall l_1, l_2.\ \text{finite}\ (l_1 +\!\!+\ l_2) \iff \text{finite}\ l_1 \land \text{finite}\ l_2$

- $\forall l.\ [] +\!\!+\ l = l$
- $\forall h, t, l.\ (h : t) +\!\!+\ l = h : (t +\!\!+\ l)$
- $\forall l.\ l +\!\!+\ [] = l$
- $\forall l_1, l_2, l_3.\ l_1 +\!\!+\ (l_2 +\!\!+\ l_3) = (l_1 +\!\!+\ l_2) +\!\!+\ l_3$

These would be previously proved properties in the library.

Applying Metis

We are now in a position to apply the Metis 'automatic' prover:

Shell

```
$ ./metis rev_rev.tptp
----------------------------------------------------------------------------
Problem: rev_rev.tptp

Goal:
finite [] ∧ (!H T. finite (H : T) <=> finite T) ∧
(!L1 L2. finite (L1 ++ L2) <=> finite L1 ∧ finite L2) ∧
(!L. [] ++ L = L) ∧ (!H T L. (H : T) ++ L = H : T ++ L) ∧
rev [] = [] ∧ (!H T. rev (H : T) = rev T ++ H : []) ∧
(!L. L ++ [] = L) ∧ (!L1 L2 L3. L1 ++ L2 ++ L3 = (L1 ++ L2) ++ L3) ==>
(!K. finite K ==> rev (rev [] ++ K) = rev K ++ []) ∧
!T.
  finite T ==> (!K. finite K ==> rev (rev T ++ K) = rev K ++ T) ==>
  !H K. finite K ==> rev (rev (H : T) ++ K) = rev K ++ H : T

Size: 19 clauses, 34 literals, 149 symbols, 149 typed symbols.

Category: non-propositional, equality, non-horn.

SZS status Theorem for rev_rev.tptp
```

Term Syntax

Here is the BNF for the *terms* of first order logic:

$$\begin{array}{rcl} \text{Term} & \leftarrow & \text{Var} \\ & | & f(\text{Term}_1, \ldots, \text{Term}_m) \end{array}$$

Terms with no variables are called *ground terms*.

Formula Syntax

And now the *formulas*:

$$
\begin{array}{lcll}
\text{Formula} & \leftarrow & \text{True} & \\
 & \mid & \text{False} & \\
 & \mid & p(\text{Term}_1, \ldots, \text{Term}_n) & /*\ \textit{atoms}\ */ \\
 & \mid & \neg\text{Formula} & \\
 & \mid & \text{Formula} \ \wedge \ \text{Formula} & \\
 & \mid & \text{Formula} \ \vee \ \text{Formula} & \\
 & \mid & \text{Formula} \implies \text{Formula} & \\
 & \mid & \text{Formula} \iff \text{Formula} & \\
 & \mid & \forall \text{Var. Formula} & \\
 & \mid & \exists \text{Var. Formula} &
\end{array}
$$

As usual, a *closed formula* is one with no free variables.

Signatures and Interpretations

- A first order *signature* is a set of possible function f/m and predicate p/n symbols (together with their arity).
- An *interpretation* of a signature is a pair (D, I).
  - D is any non-empty set, called the *domain* of elements.
  - I maps the functions and predicate symbols in the signature to domain functions and predicates:

$$I(f/m) : D^m \to D \qquad I(p/n) : D^n \to \mathbb{B}$$

- Special case: First order logic with equality always interprets the equality predicate symbol $(=/2)$ to be the equality relation on the domain.

Semantics

- Given a fixed interpretation, every (closed) formula either evaluates to true or false.
- An interpretation that makes a formula true is called a model.
  - A formula with no models is called *unsatisfiable*.
  - A formula with some models is called *satisfiable*.
  - If every interpretation is a model, the formula is called a *tautology*.
- In verification we normally have a correctness formula that we'd like to prove is a tautology.

The Bad News: Undecidability

There is no algorithm to decide whether a first order logic formula is a tautology:

$$
\begin{array}{ll}
(\forall x, y.\ ((\mathsf{k}\ .\ x)\ .\ y) \rightarrow x) & \wedge \\
(\forall x, y, z.\ (((\mathsf{s}\ .\ x)\ .\ y)\ .\ z) \rightarrow (x\ .\ z)\ .\ (y\ .\ z)) & \wedge \\
(\forall x, x', y.\ x \rightarrow x' \implies (x\ .\ y) \rightarrow (x'\ .\ y)) & \wedge \\
(\forall x, y, y'.\ y \rightarrow y' \implies (x\ .\ y) \rightarrow (x\ .\ y')) & \wedge \\
(\forall x.\ (\neg \exists y.\ x \rightarrow y) \implies \text{terminating } x) & \wedge \\
(\forall x, y.\ x \rightarrow y \wedge \text{terminating } y \implies \text{terminating } x) & \\
\implies & \\
\text{terminating (big s and k expression)} &
\end{array}
$$

The slightly better news: The problem is semi-decidable.

Metis Overview

Metis uses the following program to compute whether a closed formula F is a tautology:

1. Convert $\neg F$ to an equi-satisfiable set of clauses.
2. Deduce more clauses from the current set until one of the following conditions is met:
   - If the empty clause (i.e., False) is ever deduced, then $\neg F$ is unsatisfiable. Report that F is a tautology and terminate.
   - If no new clauses can be deduced, then $\neg F$ is satisfiable. Report that F is not a tautology and terminate.

Because the problem is semi-decidable, we know there are non-tautologies that will cause the program to loop forever.

Normalization to Clauses

- A *clause* is a disjunction of literals: $L_1 \vee \cdots \vee L_n$.
  - A *literal* is either an atom or a negation of an atom.
- How to convert an an arbitrary formula to an equi-satisfiable set of clauses?
  1. Convert all logical operations to $\neg$, $\vee$ and $\wedge$.
  2. Push the $\neg$ operations to the leaves.
  3. Lift the $\exists$ and $\forall$ quantifiers to the top.
  4. Push the $\vee$ operations beneath the $\wedge$.
  5. Introduce Skolem constants to eliminate the $\exists$ quantifiers.
  6. Drop the $\forall$ quantifiers and $\wedge$ operations.
- Introducing formula definitions avoids exponential blow-up in Steps 1 and 4.

Search Space

- The search space is all the possible clauses that can be deduced.
- However, it is not necessary to deduce all clauses, just enough to generate a proof (if there is one).
- Example: It is never useful to keep tautologous clauses

$$L \lor \neg L \lor C$$

- Warning: It is valid for a search space reduction strategy to eliminate all short proofs, so long as one proof is still reachable.

Knuth-Bendix Term Ordering

- Term orderings are commonly used to reduce the search space.
- A *term ordering* $\preceq$ is a well-founded total order on ground terms, such that if $s \preceq t$ then t is not a strict subterm of s.
- Note: If s or t contain variables, there might be grounding instantiations σ_1 and σ_2 with

$$s\sigma_1 \preceq t\sigma_1 \qquad t\sigma_2 \preceq s\sigma_2$$

- Metis uses the Knuth-Bendix term ordering, which essentially just counts the number of symbols in the term.

Ordered Resolution

In the beginning there was enumeration of terms. In 1965 Robinson introduced the resolution rule, which uses unification to combine clauses.

Inference Rule (Resolution)

$$\frac{C \vee A \qquad D \vee \neg B}{C\sigma \vee D\sigma}$$

where

1. $\sigma = mgu(A, B)$.
2. $L\sigma \preceq A\sigma$ *is satisfiable for every literal* L *in* $C \cup D$.

Ordered Factoring

Resolution is not complete without factoring.

Inference Rule (Factoring)

$$\frac{C \vee A \vee B}{C\sigma \vee A\sigma}$$

where

1. $\sigma = mgu(A, B)$.
2. $L\sigma \preceq A\sigma$ *is satisfiable for every literal* L *in* C.

Ordered Paramodulation

There is a special rule for equality.

Inference Rule (Paramodulation)

$$\frac{C \vee s = t \qquad D \vee A}{C\sigma \vee D\sigma \vee A[t]_p\sigma}$$

where

1. $A|_p$ *is not a variable.*
2. $\sigma = mgu(s, A|_p)$.
3. $s\sigma \neq t\sigma$.
4. $t\sigma \preceq s\sigma$ *is satisfiable.*
5. $L\sigma \preceq (s = t)\sigma$ *is satisfiable for every literal* L *in* C.
6. $L\sigma \preceq A\sigma$ *is satisfiable for every literal* L *in* D.

Simplification

A big advantage of using a term ordering is that we can simplify clauses and completely throw away the original.

Inference Rule (Simplification)

$$\frac{\mathcal{C} \ \cup \ \{s = t\} \ \cup \ \{C \vee A\}}{\mathcal{C} \ \cup \ \{s = t\} \ \cup \ \{C \vee A[t\sigma]_p\}}$$

where

1. *σ is the result of matching s to $A|_p$.*
2. *$s\sigma \neq t\sigma$.*
3. *$t\sigma \preceq s\sigma$ is valid.*

Example: $\forall x.\ x + 0 = x$ will always be used to simplify clauses.

Main Loop

- Maintain an active and passive set of clauses.
- Initialize the active set to be empty, and the passive set to contain the initial clauses.
- On every iteration of the main loop:
  1. Take one clause out of the passive set.
  2. Add a copy of the clause to the active set.
  3. Rename the clause with fresh variables.
  4. Combine the clause with every clause in the active set.
  5. Simplify and factor all the newly deduced clauses, and add them to the passive set.
- Invariant: All pairs of clauses in the active set have been combined.

Completeness

- Metis implements a complete search strategy, meaning that in principle it will find a proof for every tautology.
- It is reasonable to ask why it bothers, since in practice it will fail to find many proofs.
    1. When deployed in an interactive prover, it is annoying to users if a tactic cannot prove an easy goal.
    2. Metis can attempt to discover non-tautologies, because if it fails to find a proof then the formula is definitely not a tautology.

Indexing

- The main loop consists of combining one clause with a set of clauses.
- The active set maintains literal and term indexes to quickly locate all necessary unifications and matches.
- Indexing makes a big difference to performance.
  - Metis implements discrimination trees.
  - Vampire implements code trees.

Picking Passive Clauses

- Probably the important heuristic in a prover like Metis is deciding the order in which to pick clauses out of the passive set.
- Metis weights clauses when they are added to the passive set, and picks the lightest clause on every iteration.
- Clauses are given a heavier weight:
    - The later they are deduced.
    - The greater the number of literals they contain.
    - The greater the number of symbols they contain.

Logical Kernel

- Metis implements an LCF-style logical kernel.
- All the rules in the logical calculus for inferring clauses are expanded into combinations of 6 primitive inference rules.
- For example, the primitive inference rule for axioms:

$$\frac{}{C}\ \text{axiom}\ C$$

- This simplifies any kind of proof processing, because only the the primitive inferences rules need to be handled.
- For example, this is used when translating first order refutations to higher order logic proofs.

Logical Kernel: Resolution

- Excluded Middle:

$$\frac{}{L \vee \neg L} \text{ assume } L$$

- Substitution:

$$\frac{C}{C[\sigma]} \text{ subst } \sigma$$

- Resolution:

$$\frac{L \vee C \qquad \neg L \vee D}{C \vee D} \text{ resolve } L$$

  where the literal L must occur in the first theorem, and the literal $\neg L$ must occur in the second theorem.

Logical Kernel: Equality

- Reflexivity:

$$\frac{}{t = t}\ \text{refl}\ t$$

- Equality:

$$\frac{}{s \neq t \lor \neg L \lor L'}\ \text{equality}\ L\ \rho\ t$$

where s is the subterm of L at path ρ, and L' is L with the subterm at path ρ being replaced by t.

Semantic Guidance

- Following work by John Slaney, I am currently investigating using semantic properties of clauses to weight them.
- Given an interpretation with a finite domain, clauses can be evaluated as true or false.
- Clauses that are true in the interpretation are less likely to be helpful in generating an empty clause (combining two true clauses must generate another true clause).
- The main difficulty is finding a good interpretation to guide the proof search.
- So far one negative result: random interpretations don't help!

Metis: The Tool

- Written in Standard ML in a ‘purely-functional style’.
    - No destructive rewriting here.
    - Designed to emphasize clarity over performance.
    - 11,000 lines of code (+ 2,500 comment + 3,500 blank).
    - For best results use the MLton whole-program compiler.
- Available for download.
    - `http://www.gilith.com/software/metis`
    - GPL licence: hack at will, patches gratefully received.

Strengths

- The clear implementation allows Metis to be easily modded to add new kinds of automated reasoning.
- Metis reads problems in TPTP format and outputs proofs in TSTP format, so can be easily hooked up to other tools.
- Metis proofs are easily checkable, consisting of many tiny inference steps.
  - The `rev_rev` proof has 86 steps!
  - This is a result of its LCF-style logical kernel.

Weaknesses

- The more general the logic, the worse the automation.

$$\text{SAT} < \text{SMT} < \text{First Order} < \text{Higher Order} < \text{ZFC}$$

- Metis tends to be chaotic: small changes to the input can affect whether a proof is found.
  - Good for speculative background proving or easy proofs.
- Metis is not the most powerful first order prover on the market.

TPTP and CASC

- TPTP = Thousands of Problems for Theorem Provers.
- CASC = CADE Automated System Competition.
- Metis made its debut at CASC in 2007.
- It placed 10 out of 13 provers in the FOF (First Order Formula) division, proving 117 out of 300 problems (the winner, Vampire 9.0, proved 270 problems).
- Unexpected result: It solved 28 problems just in its normalization to clauses!

Deployments

- Metis is used as a proof engine in the HOL interactive theorem prover. The *METIS_TAC* tactic handles the conversion between higher order logic and first order logic.
- Larry Paulson has made good use of Metis' ability to generate explicit proofs in the Isabelle *sledgehammer* tactic, which attempts to prove your goals in a background process.
- Larry Paulson has a separate project hacking Metis to use it as an inference engine to solve problems in real closed fields.
- Geoff Sutcliffe is making use of Metis' explicit proofs and compliance to standards to extract information from proofs.

Summary

- This talk has presented the Metis first order prover.
- The 40+ years of work on first order provers have generated quite a bit of background theory.
- And yet despite this, some interesting research problems still remain to make them more robust and powerful.