Theorem Provers Part 2: Inefficient Computation

Thomas Sewell

Dec 2024

Theorem Provers & Reflected Computation

In part 4 of this lecture series, we'll look at proof by reflection: yet another take on proofs as computer programs.

Some relevant ingredients to recall:

- From day 2, problems that can be inefficient to prove:

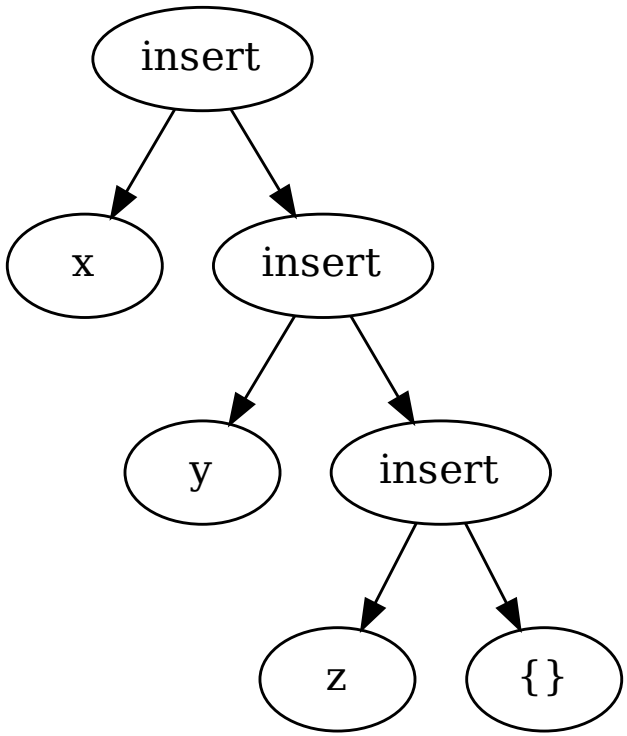
$$\{x, 3, 4, 5, 6\} = \{6, 5, 4, 3, x\}$$

- From day 3, that we aim to validate tedious/exhaustive proofs.
- From day 3 (& week 1), that we can prove things about programs.

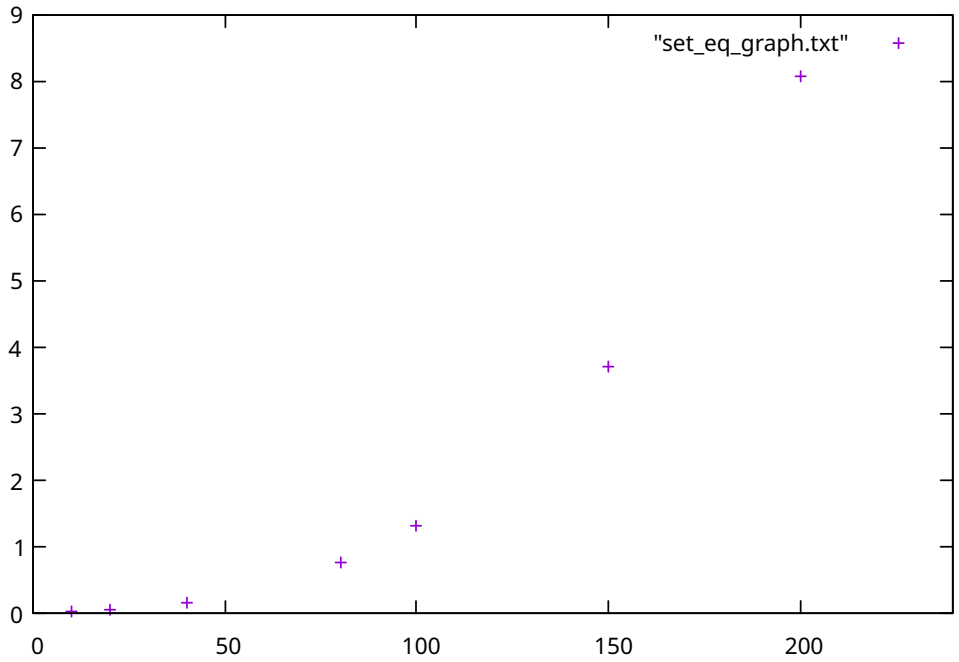
Back to Set Equality

To revisit our set equality problem, the issue is that the syntax is linear.

$$\{x, 3, 4, 5, 6\} = \{6, 5, 4, 3, x\}$$



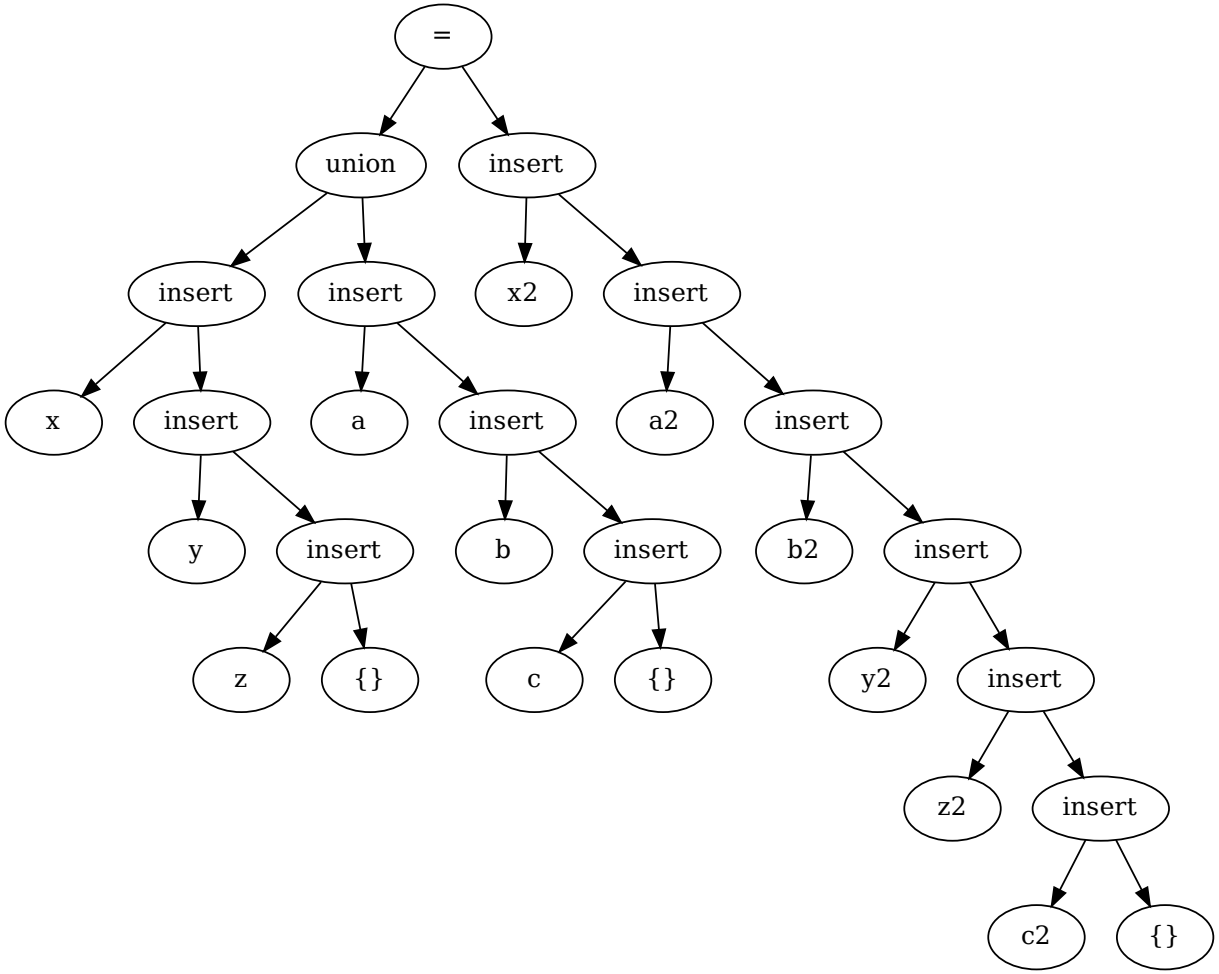
Back to Set Equality



Our graph is (a) quadratic and (b) the numbers are small.

Adding Unions

$$\{x, y, z\} \cup \{a, b, c\} = \{x, a, b, y, z, c\}$$



Sorting via Union Split/Merge

$$\{x, a, b, y, z, c\} = \{x, y, z\} \cup \{a, b, c\}$$

We can write a proof tool based on this strategy. Given a set construction:

1. Sort the syntax (in our implementation language).
2. If the construction is in order, stop.
3. Divide the sorted syntax into “front” half and “back” half.
4. Propose a division into the two halves, and prove it in $O(n)$ steps.
5. Recurse on the halves.

This is about the best we can do in Isabelle/HOL.

Pros and Cons

Strengths of this (kind of) approach:

- We can just do it (in an LCF-style tool).
- Complexity is $\approx O(n \log(n))$ proof steps.

Limitations:

- The algorithm is fiddly and specialised.
- Complexity is $\approx O(n \log(n)^2)$ in the implementation.

Pros and Cons

Strengths of this (kind of) approach:

- We can just do it (in an LCF-style tool).
- Complexity is $\approx O(n \log(n))$ proof steps.

Limitations:

- The algorithm is fiddly and specialised.
- Complexity is $\approx O(n \log(n)^2)$ in the implementation.
- Can we do more in "native" computation?

Pros and Cons

Strengths of this (kind of) approach:

- We can just do it (in an LCF-style tool).
- Complexity is $\approx O(n \log(n))$ proof steps.

Limitations:

- The algorithm is fiddly and specialised.
- Complexity is $\approx O(n \log(n)^2)$ in the implementation.
- Can we do more in "native" computation?

Is this useful in practice? I have seen it most often involving language infrastructure.

Computations in Logic

For **pure, functional, terminating** programs, their meaning is the same in the logic as in computation.

For instance `map ::` $(\alpha$ `->` $\beta)$ `->` α `list ->` β `list`

- `map f [] = []`
- `map f (x : xs) = f x : map f xs`

Not true for impure functions (e.g. `getEnv :: string -> string`) or for potentially nonterminating functions. In these cases, the logical/mathematical type needs to acknowledge the potential effects.

Doing Computation in Logic

Given `f :: (`α `-> `β`)`, `x :: `α, how can we prove that `f x = y`?

We can write an evaluator using ordinary proof tools in the logic.

We could also just evaluate `f x` in a conventional programming language environment.

- Typically orders of magnitude faster.
  - In principle it is a constant factor.
- Adds the language environment to the trusted core.

Doing Computation in Logic

Given `f :: (`α `-> `β`)`, `x ::` α, how can we prove that `f x = y`?

We can write an evaluator using ordinary proof tools in the logic.

We could also just evaluate `f x` in a conventional programming language environment.

- Typically orders of magnitude faster.
  - In principle it is a constant factor.
- Adds the language environment to the trusted core.

Most intuitionistic tools have such an interpreter. Most HOL implementations do not. ACL2 exposes (a safe subset of) the host Lisp implementation.

Appealing to a Logic

We have also seen:

- How to characterise a logic.
- How to prove a logic L_1 sound (in L_2).
- How (in principle) to implement a logic as a program.

So, given a goal G in our host theorem prover

- phrase an equivalent goal G_1 in an inner logic L_1
- run `checker` G_1 and get `True`
- conclude from the correctness of `checker` that $L_1 \vdash G_1$
- from the soundness of L_1 that G

Deep and Shallow Encodings

To use this trick, we need to construct a **deep encoding**.

This is a standard distinction when we encode a logic or language in another. Do we map expressions to their syntax or their semantics?

- Is 1 + 1 the same thing as 2?
- Is $\mathit{True} \wedge \mathit{True}$ the same thing as True?
- Does $\{1, 2, 3\}$ have a first element?

Deep and Shallow Encodings

To use this trick, we need to construct a **deep encoding**.

This is a standard distinction when we encode a logic or language in another. Do we map expressions to their syntax or their semantics?

- Is 1 + 1 the same thing as 2?
- Is *True* $\wedge$ *True* the same thing as *True*?
- Does $\{1, 2, 3\}$ have a first element?

```
datatype α Set_Encoding = Insert α (α Set_Encoding)
    | Empty

fun interp ::  (α -> β) -> α Set_Encoding -> β Set
    where
        interp f Empty = {}
        interp f (Insert x s) = ({f x} ∪ interp s)
```

Deep and Shallow Encodings and Reflection

$$\{x, 3, 4, 5, 6\} = \{6, 5, 4, 3, x\}$$

```
datatype Number_Encoding = Numeral Int | Variable String
fun interp2 ::  (String -> Int) -> Number_Encoding -> Int
    where
        interp2 env (Numeral i) = i
        interp2 env (Variable nm) = env nm
fun interp3 ::  (String -> Int) ->
                Number_Encoding Set_Encoding -> Int Set
    where
        interp3 env enc = interp (interp2 env) enc
```

```
fun interp3 ::  (String -> Int) ->
                Number_Encoding Set_Encoding -> Int Set
```

To make use of this, we need a gadget that looks at our goal G, and invents an encoding G_{env} and G_{enc} such that `interp3` G_{env} G_{enc} = G.

This is a **reflection** of part of the logic back into itself.

We can then manipulate this deep encoding, and reason about the interpretation of these manipulations.

Reflected Term Normalisation in Coq

$$\{x, 3, 4, 5, 6\} = \{6, 5, 4, 3, x\}$$

I used this puzzle and its relatives to help learn Isabelle, HOL4, ACL2 and Coq.

In Coq:

- Define a type of deep encodings of relevant syntax.
- Map the goal G to a deep reflection of itself G_{enc}.
- Define an ordinary sorting/normalising pass `f`.
- Verify `f`, i.e. show $\forall syn.$ `interp (f` syn`)` $\vdash$ `interp` syn.
- Check that `interp` $G_{enc} = G$.

Reflected Term Normalisation in Coq

$$\{x, 3, 4, 5, 6\} = \{6, 5, 4, 3, x\}$$

I used this puzzle and its relatives to help learn Isabelle, HOL4, ACL2 and Coq.

In Coq:

- Define a type of deep encodings of relevant syntax.
- Map the goal G to a deep reflection of itself G_{enc}.
- Define an ordinary sorting/normalising pass `f`.
- Verify `f`, i.e. show $\forall syn.$ `interp (f` syn`)` $\vdash$ `interp` syn.
- Check that `interp` $G_{enc} = G$.

It's neat that this works! But, the reflection part was disappointing.

Reflected Meta-Strategies in ACL2

$$\{x, 3, 4, 5, 6\} = \{6, 5, 4, 3, x\}$$

In ACL2:

- Every term is in the type of Lisp trees.
  - `('x, 3, 4, 5, 6)` & `(car, cdr)`
- Define a syntactic criterion for relevant syntax.
  - ACL2 synthesises an equivalent of `interp`.
- Define an ordinary Lisp sorting/normalising pass `f`.
- Verify `f`, i.e. show $\forall x$. `f` $x \neq$ `nil --> interp (f` x`) = interp` x.
- Add `f` as a meta-method.

Performance-wise, this works really well! But it's Lisp.

More Reflected Logic?

We've looked at a silly exercise for reflection.

What about running a general hosted logic implementation?

- A tableau solver?
- A SAT solver?
- An implementation of a general-purpose logic?

Recap

Recap. How can we efficiently prove a goal?

- Assume a computational escape-hatch for `f x = y`.
- Reflect goal G into an encoding G_{enc}.
- Run a relevant/efficient proof from some logical prover L.
- Appeal to the soundness of L.

That's all for today. Thanks for listening.

Questions?