Functional Pearl: DIY Elaborator for Internal Solvers Using Time Travel, Transformers, Free Scoped Monads and Term Reconstruction

1 INTRODUCTION

Being able to formally verify a complex mathematical theorem or to prove that a critical piece of software runs correctly is always a delight. Due to the formal nature of the task, however, this requires lots of technical lemmas to be proven, which are nigh trivial for a human mind but which computer might struggle to solve automatically. Consider the following lemma from Lean's `mathlib4` (in `Algebra.Group.Basic`):

```
variable [CommSemigroup G]
theorem mul_right_comm (a b c : G) : a * b * c = a * c * b := by
  rw [mul_assoc, mul_comm b, mul_assoc]
```

This code snippet states and proves that, in any commutative semigroup G, given three elements $a, b, c \in G$, it is true that $a \cdot b \cdot c = a \cdot c \cdot b$. To mathematically educated person, the correctness of the lemma itself is obvious; however, the one who wrote the proof needed three steps of rewriting the type (by associativity of multiplication, by commutativity and then by associativity again) to convince the typechecker.

It would surely be nice to avoid this chore! To aid us all, proof assistant designers usually provide the means to automate the generation of proofs for trivial lemmata. One viable approach is to use SMT solvers which, given a statement formulated in one of the more restricted logical theories, decide whether the statement is true and output the proof. Usually, such solvers are written in another programming language or use metaprogramming facilities.

In this paper, we contribute in the following ways:

- In Section 2 we sketch a technique which enables a user of a proof assistant to write solvers for decidable theories inside proof assistant and immediately use them in this same system without any macros.
- To back up our claims, in Section 3 we describe a suitable minimal type system and in Section 4 we provide a proof-of-concept elaboration algorithm written in Haskell.
- By composing well-known solutions from pure functional programming world and making use of Haskell's laziness, in Section 4 we arrive at the description of an elaboration algorithm as a single-pass AST traversal.

2 SKETCH OF A SOLVER

Essentially, our technique relies on two tricks:

(1) Choosing a nice type for a solver function written in a prover language;
(2) Structuring typing rules in such a way so that most of the arguments to said function can be inferred (this task is explored in more detail in Section 3).

The task seems circular: we need to specify the type for a solver function, so we need to fix the type system; but to specify typing rules, we need to know the type of the solver to design against. Lucky for us, the type of a solver is expressible in any type system featuring the following:

(1) The empty type $\bot$;
(2) The disjoint union (or a sum type) $A + B$;
(3) The type of dependent functions $(x : A) \to B(x)$ (in the case where B does not depend on x, we write $A \to B$);
(4) The type of dependent pairs $(x : A) \times B(x)$ (in the case where B does not depend on x, we write $A \times B$);
(5) Ability to form inductive datatypes, including but not limited to:
    - The type of natural numbers $\mathbb{N}$;
    - For each $n : \mathbb{N}$, the type of natural numbers less than n, denoted $\underline{n}$.
(6) The type of propositions **prop**;
(7) The type of small types **type**;
(8) The propositional equality type $a =_A b$.

As an example, let's consider the construction of a solver type for equational theory of commutative (multiplicative) semigroups, that is, a theory where:

- We can ask questions about equalities of finite expressions written using infix multiplication $(\cdot)$ and finite number of variables $x_1, \ldots, x_n$.
- Two expressions are considered equal if we can get one from another applying a finite sequence of rewrites of two kinds: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ and $a \cdot b = b \cdot a$.

From user's point of view, such solvers are directly applicable to automatically prove statements as the one used in the example from Section 1, $(a \cdot b) \cdot c = (a \cdot c) \cdot b$.

Then, in a type system having features outlined above, the following terms can be defined:

(1) A type of *expressions* $E : (n : \mathbb{N}) \to \textbf{type}$ in our theory is a type of binary trees where nodes denote multiplication and leaves store variable indices taken from $\underline{n}$;
(2) A *formula* F with n variables is a pair of expressions (left-hand side of an equation, right-hand side):

$$F(n : \mathbb{N}) \coloneqq (E\ n) \times (E\ n);$$

(3) A *model* $\mathcal{M}$ with carrier X is a dictionary of functions and properties which turn a type X into a semigroup:

$$\mathcal{M}(X : \textbf{type}) \coloneqq ((\cdot) : X \to X \to X) \times ((a\ b\ c : X) \to a \cdot (b \cdot c) = (a \cdot b) \cdot c) \times ((a\ b : X) \to a \cdot b = b \cdot a);$$

(4) An *embedding function* P which interprets formulas of a theory as internal propositions:

$$P : (n : \mathbb{N}) \to (X : \textbf{type}) \to (M : \mathcal{M}\ X) \to (C : \underline{n} \to X) \to F\ n \to \textbf{prop};$$

(5) An *internal proof* Prf of a formula ϕ is a proof that it holds universally, over all models and interpretations:

$$\mathrm{Prf}(n : \mathbb{N})(\phi : F\ n) \coloneqq (X : \textbf{type}) \to (M : \mathcal{M}\ X) \to (C : \underline{n} \to X) \to P\ n\ X\ M\ C\ \phi$$

(6) A *decision function* $\mathcal{D}$ which, in fact, expresses the logic of a solver – here it's enough to compare "normal forms" of two expressions by counting number of occurences of each variable in each expression:

$$\mathcal{D} : (n : \mathbb{N}) \to (\phi : F\ n) \to (\mathrm{Prf}\ n\ \phi + (\mathrm{Prf}\ n\ \phi \to \bot));$$

(7) Finally, we would like to define the following "wrapper"

$$\mathbf{solver} : (n : \mathbb{N}) \to (\phi : F\ n) \to (\pi : \mathrm{Prf}\ n\ \phi) \to (\mathrm{eq} : \mathcal{D}\ n\ \phi =_{\dots} \mathrm{inl}\ \pi) \to$$
$$\to (X : \mathbf{type}) \to (M : \mathcal{M}\ X) \to (C : \underline{n} \to X) \to P\ n\ X\ M\ C\ \phi.$$

The result of this function can be computed as simply as $\pi\ X\ M\ C$; the secret sauce is the type. It is constructed in such a way that arguments can be provided to the **solver** iff $\mathcal{D}$ states that a formula is, in fact, provable.

In order for this framework to be practical, however, all of the wrapper's arguments (except, maybe, for eq, M and C) have to be inferrable from resulting type. For example, lemma from Section 1 would then be provable as simply as

$$\mathbf{solver}\ \_\ \_\ \_\ (\mathrm{refl}\ \_\ \_)\ \_\ M\ [a, b, c] : a \cdot b \cdot c =_{\_} a \cdot c \cdot b,$$

where refl $A\ x : x =_A x$, underscores denote terms which should be inferred by the typechecker and $[x_1, \dots, x_n]$ is a shorthand for a function from $\underline{n}$ to X mapping i to x_i. For ease of presentation, we refrain from adding further niceties to the surface syntax (such as implicit arguments and instance resolution), but they of course should be present in any practical system.

3 TYPING RULES

4 IMPLEMENTATION

5 RELATED WORK

REFERENCES