CS240H: A Standalone Proof-checker for the Lean Theorem Prover

Daniel Selsam

1 Project Overview

I implemented a reference type checker for the Lean Theorem Prover [dMKA+15]. Lean is an interactive theorem prover based on the Calculus of Inductive Contsructions (CIC), and is very similar to Coq. Lean itself is a tremendously complicated system. The user enters information in the form of highly ambiguous "pre-terms" that may require overloaded notation to be disambiguated, implicit (potentially higher-order) arguments to synthesized, coercions to be inserted, and even computation to be performed in order to construct complete terms in the core underlying logic. This process is called "elaboration" and is notoriously complex and byzantine, since the different kinds of ambiguity can interact with each other in global and unpredictable ways. Moreover, even the Lean proof-checker itself is substantially more complicated than correctness alone would require, because (a) it needs to be very performant since the user waits for it in real time, and (b) it needs to support some features that are only used by the elaborator and are irrelevant to the core underlying logic.

One of Lean's primary purposes is to check the correctness of proofs, and all the complexity discussed above makes it hard to trust that Lean indeed only certifies proofs that are actually correct. One approach to increasing the trustworthiness of the system is to export the complete proof terms to a file and have an entirely self-contained executable that implements a minimal proof checker for the logic parse the export file and re-check the proofs. This is precisely what I have implemented for this project. An additional executable that renders the complete proof terms in ways that can be more easily read by humans would also be valuable, since "un-elaborating" is a vastly more straightforward process than elaborating, though I have not implemented such an "un-elaborator".

2 System Overview

The proofs are exported into a compact format described at `https://github.com/leanprover/lean/blob/master/doc/export_format.md`. The export file contains (a) a sequence of expressions that it internally refers to by unique integer ids, (b) a collection of inductive datatype declarations, and (c) a collection of definitions which consist of a name, an expression that represents the value, and an expression that represents the claimed type of the value in question. By the Curry-Howard Isomorphism, the same "definition" machinery is used to represent both functional programs and mathematical proofs. In the latter case, the value is the proof, and the type of the value is the theorem that the proof proves. Therefore the primary task of the reference type checker is to (a) infer the types of the proofs, and (b) confirm that the inferred types are definitionally equal to the theorems that are claimed to have been proved.

2.1 Expressions

As discussed above, a Lean expression can represent (a) a functional program, (b) the type of a functional program, (c) a proof of a theorem, or (d) the statement of a theorem. We represent this expression type with a algebraic datatype in Haskell. The actual implementation puts all the constructor arguments inside constructor-specific types, so that we can let other modules pattern-match on expressions while caching some useful quantities transparently, and so has complexity that is not helpful when trying to understand what an expression is. Here is a simplified form of the expression type:

```
data Expr = Var Int
          | Local Name Expr
          | Sort Level
          | Constant Name [Level]
          | Lambda Expr Expr
          | Pi Expr Expr
          | App Expr Exp
```

1. A Var is a bound variable that is represented by its de-Bruijn index.
2. A Local is a free variable that has a unique name and carries its type. We create these when we traverse under binders, so we do not need to weave a typing context around.
3. A Sort is a type whose elements are types, and is usually denoted by the symbol Type. A Level is a "universe level", and tracking them is a necessary evil in order to avoid Russell's Paradox. We will not discuss them futher, except to say that an element of type Prop := Type_0 is a theorem statement (a.k.a. a "proposition"), and an element of type Type_k for $k > 0$ is the type of a functional program.
4. A Constant is a reference to a previously defined expression in the current environment.
5. A Lambda is just a lambda abstraction, and takes the domain type and the body type.
6. A Pi is a universal quantifier for propositions (i.e. forall), and for programs is a (dependent) function type. Like a lambda abstraction, it takes the domain type and the body type. Note that a Pi type for which the body does not depend on the binder can be thought of as an implication for propositions (e.g. $A \to B$) and as a regular function for programs. An example of a dependent function is the function that takes a natural number n to the zero-vector of length n.
7. An App is an function application of one expression (that encodes a function) to another.

2.2 Type inference

Type inference is a fairly simple recursive function, for which we do not use any particularly interesting features of Haskell, so we do not discuss it here.

2.3 Definitional equality

In CIC, unlike in FOL, expressions have computational behavior associated with them. When two terms are equal upto computational behavior, we say they are *definitionally* equal. There are many computation rules, many of which will be familiar to the reader. Here are some examples:

1. β: $(\lambda x, x + x)5$ is definitionally equal to $5 + 5$.
2. η: $(\lambda x, fx)$ is definitionally equal to f.
3. δ: If $x := y$ is a definition in the environment, then fx is definitionally equal to fy.
4. ι: If $H : A = A$ and a has type A, then eq.rec a H is definitionally equal to a (note that are are hiding some implicit arguments). There are rules like this for every inductive type, which we discuss more below.

One of the central tasks for the reference type checker is to determine if two terms are definitionally equal or not. Unfortunately, even though the computation rules are strongly normalizing, it would be too slow to normalize both expressions every time we need to check for equality. As a consequence, our definitional-equality test is much more complicated, and must lazily take computation steps and check for equality in order to avoid unnecessary computation. Moreover, we try many quick yet incomplete definitional equality checks as well. We discuss this procedure more below when we discuss the "short-circuit" monad.

2.4 Inductive datatypes

One of the most central and celebrated features of the CIC is the support for inductive types. Inductive types present a way to define a new kind of object or judgment, to specify exactly how one can construct elements of said type, and to specify what is required to define functions on or proof theorems about elements of that type. Here is an example of an inductive type in a non-dependent subset of Lean that looks like it could be Haskell:

```
inductive nat :=
| zero : nat
| succ : nat -> nat
```

This declaration gives us a new type nat as well as two ways of constructing nats, as it would in Haskell. In addition. it gives us what is called a *recursor* nat.rec for nat, which has the following form:

```
Pi (C : nat -> Type),
  C zero ->
    (Pi (a : nat), C a -> C (succ a)) ->
      (Pi (n : nat), C n)
```

The astute reader will recognize this as precisely the principle of mathematical induction for natural numbers. The recursor also tells us how to define functions that take nats as an argument.

Finally, the declaration gives us a computation rule for the recursor that does the obvious thing, to be used in ι-reduction. Specifically,

```
nat.rec Czero Csucc zero ⟹ Czero
nat.rec Czero Csucc (succ n) ⟹ Csucc n (nat.rec Czero Csucc n)
```

For example, we can define the double function as follows:

```
definition double : nat -> nat :=
  nat.rec zero (Lambda n dn, succ (succ (dn)))
```

It is the responsibility of the reference type checker to validate an inductive datatype declaration, to generate the recursor and to determine the appropriate computation rules. The general formulation of recursors for inductive types is rather complicated (and rather ingeneous—see [Dyb94] for details), and the implementation is correspondingly complex.

2.5 Evaluation

I used the reference type checker to verify all the proofs in the Lean standard library, which consists of over 10,000 definitions and almost 200 different inductive types. The entire process, including parsing, generating the recursors, inferring types and confirming definitional equality takes a little over thirty seconds on my machine. The reference type checker is sufficiently parameterizable that it can support Homotopy Type Theory (HoTT) [Awo15] as well, and I also used it to verify all the proofs in the Lean HoTT library, which consists of almost 8,000 definitions and almost 200 different inductive types. The entire process for the HoTT library takes about one minute on my machine.

3 Haskell in action

The rest of this report discusses interesting uses of Haskell in the reference type checker. First we discuss "not-so-ugly" idioms that recover abstractions standard in other languages without being unbearably ugly. Next we discuss "beautiful" idioms that qualitatively improve the simplicity and comprehensibility of parts of the program compared to how they would be implemented in more traditional languages.

3.1 Not-so-ugly Haskell

3.1.1 Simple lenses

Although an earlier version contained some egregious code to make updates inside nested structures, I refactored after learning about lenses and now these sections are rather elegant. Here is a canonical example using over:

```
envAddIntroRule irName indName = over (envIndExt . indExtIntroNameToIndName)
                                      (Map.insert irName indName)
```

The original code is too ugly to include in this report. Here is another example of a "not-so-ugly" line, inside a do statement inside a State monad computation[1]:

```
-- DS.equivalent :: Int -> Int -> (Bool, DS.IntDisjointSet)
deqCacheDS %%= DS.equivalent n1 n2
```

This line (a) assigns the second argument to the lens deqCacheDS and (b) returns the Bool result. Without lenses, one would have needed to do something much uglier like the following:

```
let (result, newCache) = DS.equivalent n1 n2
modify (\deq -> deq { deqCacheDS = newCache })
return result
```

3.1.2 Simulating objects with monad transformers

It is very useful to be able to execute some code in the context of some kind of state, some of which may be mutable and some of which may not be, and to be able to throw exceptions. This is a large part of the useful functionality provided by objects in C++. We simulate that in Haskell with monad transformers. Here is the type of "methods" on the "TypeClass" object:

```
type TCMethod = ExceptT TypeError (StateT TypeCheckerS (Reader TypeCheckerR))
```

We are essentially defining a TC class that throws exceptions of type TypeError, and has read-only state TypeCheckerR and mutable state TypeCheckerS. The monad transformers are very well designed and there are only a few points in the code where the stack is visible; most of the time all the usual functions for each of the three monad types in the stack do what we expect. The reason is that there are extra monad-like types being inferred by type class inference, such as MonadState, upon which the important functions are defined. For example, the following line uses the MonadState function uses even though the outermost monad transformer is not StateT:

```
levels <- uses (addIndIDecl . indDeclLPNames) (map mkLevelParam)
```

3.1.3 Using the ParsecT monad transformer

We want our parser to perform computations directly without needing to construct a parse tree explicitly. In an earlier design, the parser returned a ExceptT ExportError (S.State Context) a computation that would be run independently. I found this approach did not quite clear the "not-so-ugly" bar, so I refactored to make the parser return () but wrap the underlying monad directly.

```
type ParserMethod = ParsecT String () (ExceptT ExportError (S.State Context))
parseExportFile :: ParserMethod ()
```

Now each parser function parses as usual, and then lifts a computation in the underlying monad at the end. For example:

[1] Note that I recently removed the entire module that this excerpt came from, since it added unnecessary complexity.

```
parseEV = do
  newIdx <- parseInteger <* blank
  string "#EV" >> blank
  varIdx <- parseInt
  lift $ do
    use ctxExprMap >>= assertUndefined newIdx IdxExpr
    ctxExprMap %= Map.insert newIdx (mkVar varIdx)
```

This seems natural to me.

3.2 Beautiful Haskell

Both of the Haskell gems we present involve monad transformers.

3.2.1 The short-circuit monad transformer

As we discussed above, there are many ways that one can prove that two terms are definitional equality. The C++ implementation of the definitional equality checker has many methods that return an element of an enum type of size three: true, false, unknown. There are many boiler plate lines that check if the result is not unknown, and if it is not, returns the result. Here is an example from the Lean kernel in C++:

```
lbool r = quick_is_def_eq(t, s, cs, use_hash);
if (r != l_undef) return to_bcs(r == l_true, cs);

expr t_n = whnf_core(t);
expr s_n = whnf_core(s);

if (!is_eqp(t_n, t) || !is_eqp(s_n, s)) {
    r = quick_is_def_eq(t_n, s_n, cs);
    if (r != l_undef) return to_bcs(r == l_true, cs);
}

r = reduce_def_eq(t_n, s_n, cs);
if (r != l_undef) return to_bcs(r == l_true, cs);
```

In my Haskell implementation, I use the ExceptT monad transformer to allow "short-circuiting" behavior, so that the return value need not be checked every time. In particular, if a function would have returned true or false, it simply calls throwE true or throwE false accordingly, and if it would have returned unknown, then it simply returns without throwing an exception. The resulting code is much easier to follow:

```
do quickIsDefEq t s
   t_n <- lift $ whnfCore t
   s_n <- lift $ whnfCore s
   deqTryIf (t_n /= t || s_n /= s) $ quickIsDefEq t_n s_n
   (t_nn, s_nn) <- reduceDefEq t_n s_n
```

where we define the combinator

```
deqTryIf :: Bool -> DefEqMethod () -> DefEqMethod ()
deqTryIf b check = if b then check else return ()
```

Note that the lift is necessary to lift non-short-circuiting computations into the "short-circuit" monad. I wrote many other useful combinators as well, such as

```
deqCommitTo :: DefEqMethod () -> DefEqMethod ()
deqCommitTo deqFn = deqFn >> throwE False

deqTryAnd :: [DefEqMethod ()] -> DefEqMethod ()
deqTryAnd [] = throwE True
deqTryAnd (deqFn:deqFns) = do
  success <- lift $ runExceptT deqFn
  case success of
    Left True -> deqTryAnd deqFns
    _ -> return ()

deqTryOr :: [DefEqMethod ()] -> DefEqMethod ()
deqTryOr [] = return ()
deqTryOr (deqFn:deqFns) = do
  success <- lift $ runExceptT deqFn
  case success of
    Left True -> throwE True
    _ -> deqTryOr deqFns
```

and as a result, the code is much cleaner and easier to follow.

3.2.2 The MaybeT monad transformer

A similar issue comes up when, if any of a sequence of computations fails to produce a result, the entire function fails to produce a result. This functionality is usually acheived in traditional languages by checking and possibly failing after each such function call, but in Haskell can be captured elegantly using the MaybeT transformer.

The following is a (slightly simplified) excerpt from the Lean kernel in C++:

```
if (!is_constant(app_type_I) ||
    const_name(app_type_I) != it->m_inductive_name)
    return none_ecs();
auto new_intro_app = mk_nullary_intro(env, app_type, it->m_num_params);
if (!new_intro_app)
    return none_ecs();
expr new_type    = ctx.infer_type(*new_intro_app);
if (has_expr_metavar(new_type))
    return none_ecs();
if (!ctx.is_def_eq(app_type, new_type))
    return none_ecs();
return some_ecs(*new_intro_app);
```

which we can write in Haskell as follows:

```
appTypeOpConst <- liftMaybe $ maybeConstant appTypeOp
guard (constName appTypeOpConst == elimInfoIndName einfo)
newIntroApp <- liftMaybe (mkNullaryIntro env
                                         appType
                                         (elimInfoNumParams einfo))
newType <- lift $ inferType newIntroApp
(lift $ isDefEq appType newType) >>= guard
return newIntroApp
```

References

[Awo15] Steve Awodey. Homotopy type theory. In *Logic and Its Applications*, pages 1–10. Springer, 2015.

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction-CADE-25*, pages 378–388. Springer, 2015.

[Dyb94] Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.