Proof Reconstruction: Parsing Proofs Project Report

Diego Alejandro Montoya-Zapata

EAFIT University, Colombia,
dmonto39@eafit.edu.co

1 Problem Statement

Agda is a proof assistant. It is an interactive system for writing and checking proofs. Agda is also a functional language with dependent types (Bove, Dybjer, & Norell, 2009). Dependent types are simply types that depend on other types. For example, we can say that V^n, that represent the vectors of numbers of length n, is a dependent type, due to V^n depends on n, where every n is an element of the type $\mathbb{N}$, that represent the non-negative integers (Bove & Dybjer, 2009).

"Automated Theorem Proving (ATP) deals with the development of computer programs that show that some statement (the conjecture) is a logical consequence of a set of statements (the axioms)" (Sutcliffe, Zimmer, & Schulz, 2004).

TPTP is a language understood by most of the ATPs, that is, TPTP is a language for writing ATP problems. In the other hand, TSTP is a language for the proofs performed by ATPs then, this is a language for writing solutions to ATP problems (Sutcliffe et al., 2004). It is important to mention here that not all the ATPs print their proofs in TSTP format.

Currently, Sicard-Ramírez is working on a new approach in computer-assisted verification of lazy functional programs. He provided a translation of the Agda representation of first-order formula into TPTP.This translation was performed by the Apia program (Sicard-Ramírez, 2015).

In this moment, ATPs are being used as oracles, that is, a formula in first-order logic is sent to ATPs, via the Apia program, and these try to find a proof for the argument. If a proof is found, we assume that it is right and, thus, the theorem has been proved. However, this is unacceptable for most Agda users, since they consider a theorem proved only if it has been verified by the proof assistant and the user must also trust that translation from Apia into ATPs logic is being done adequately. Thus, in order to increase the reliability of this process, it would be highly desirable to establish a communication in the other direction, by the reconstruction of the Agda proof associated with the proved conjectures, from their ATPs proofs.

2 Goal

The TPTP library has provided the community with standards for input and output for ATPs (Sutcliffe, 2009). However, it does not exist a standard for the way the proof is printed, which make it difficult to try to do a program to reconstruct the proofs for all of the ATPs. For this reason, we decided to focus our efforts in formulating the demonstration in Agda just for one ATP.

3 Work Done

In this section, some of the advances of the project are shown as well as some of the troubles had during the development of the project are also described.

First, due to software restrictions, SPASS was chosen as the ATP to perform the reconstruction of its proofs (Weidenbach et al., 2009). Nevertheless, it had to be changed, since the output for the proofs from this ATP was not TSTP and it was not possible to get the most recent version of the ATP. Finally, the decision was to pick E as the ATP for performing the proof reconstruction (Schulz, 2013).

In Figure 1 is shown a piece of the *modus ponens* proof in E. The proof is in TSTP format.

```
fof(c_0_7, plain, ((~a|b)), inference(fof_nnf,[status(thm)],
                                                [c_0_4])).
cnf(c_0_10,plain,(b|~a), inference(split_conjunct,
                                   [status(thm)],[c_0_7])).
cnf(c_0_13,plain,(b|~a), c_0_10).
cnf(c_0_14,plain,(a), c_0_11).
cnf(c_0_16,plain,(b), inference(cn,[status(thm)],[inference
     (rw,[status(thm)],[c_0_13, c_0_14, theory(equality)]),
                              theory(equality,[symmetry])])).
```

Figure 1: Proof of the *modus ponens* principle in E.

Then, before starting with the proof reconstruction, I decided to focus in the *prerequisites*, that is, I decided to firm up my knowledge about Haskell and Agda, since these are the basic tools to be used for the succeed of this project.

Regarding to Haskell, the main efforts were focus in answering a question: *How to parse in Haskell?* I read several tutorials tutorials, some of them were *Combinator Parsing: A Short Tutorial* (Swierstra, S. Doaitse, 2009), and *Parsing Log Files in Haskell*[1].

A code in Haskell for parsing an IP address[4], using *attoparsec*, is shown in Figure 2. Notice that an IP address has the following format, where each number is in the range from 0 to 255.

```
<number>.<number>.<number>.<number>
```

[1] https://www.fpcomplete.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/attoparsec

```
data IP = IP Word8 Word8 Word8 Word8

parseIP :: Parser IP
parseIP = do
  d1 <- decimal
  char '.'
  d2 <- decimal
  char '.'
  d3 <- decimal
  char '.'
  d4 <- decimal
  return ( IP d1 d2 d3 d4 )
```

Figure 2: Code for parsing an IP address.

Regarding to Agda, it was necessary to do a deeper work. First, I read *Dependent Types at Work* (Bove & Dybjer, 2009). It was a good and useful introduction to Agda. I show in Figure 3 a naive definition for *equality* among natural numbers, using the data types `Nat` and `Bool` defined in (Bove & Dybjer, 2009).

```
data Bool : Set where
  true  : Bool
  false : Bool

data Nat : Set where
  zero : Nat
  succ : Nat -> Nat

_==_ : Nat -> Nat -> Bool
zero   == zero   = true
zero   == succ n = false
succ n == zero   = false
succ n == succ m = n == m
```

Figure 3: Definition for equality between natural numbers.

Moreover, due to the proofs given by the ATPs are performed by contradiction, it was necessary to learn how to perform proofs by contradiction in Agda. First, it is needed to take into account that Agda does not work by default with the of indirect proof principle so, it has to be defined. In Figure 4, you can see a definition of this principle, made by Sicard-Ramírez[2] and in Figure 5 is the postulation of the principle of the excluded middle *pem*, which was necessary for the defintion of the principle of indirect proof[3].

[2] `https://github.com/asr/agda-ptrlib/blob/master/src/Properties.agda`
[3] `https://github.com/asr/agda-ptrlib/blob/master/src/Base.agda`

```
pip : {A : Set} → (¬ A → ⊥) → A
pip h = case (λ a → a) (λ ¬a → ⊥-elim (h ¬a)) pem
```

Figure 4: The indirect proof principle, for performing proofs by contradiction in Agda.

```
postulate pem : ∀ {A} → A ∨ ¬A
```

Figure 5: Postulation of the principle of the excluded middle in Agda.

After defining the indirect proof principle, it was necessary to learn how to use it and be able to perform proofs by contradiction in Agda. In Figure 6 you can find an example of the *modus ponens* proof by contradiction performed in Agda (Da Costa & de Ronde, 2014).

```
ppi→←' : {A : Set} → A → A
ppi→←' = pip (λ h → h (λ a → a))
```

Figure 6: Proof of the *modus ponens* principle by contradiction in Agda.

During the last weeks we worked in parsing from TSTP into Haskell. So, we looked for information about the rules of inference used by E, since this is the basis for the reconstruction of the proofs.

4 Work in Progress

Currently, we are testing the behavior of the parser from TSTP into Agda, developed by Gómez-Londoño.[4] We are now verifying its performance with basic E proofs, in order to start with the core of the project.

What we have in this moment is the basis for the construction of a kind of DAG (Directed Acyclic Graph), where each node represent a step of the proof. Moreover, each node contains information about who its parents are, namely, each step contains the information of how it was inflicted.

5 Future Work

It is necessary to start with the reconstruction of the proof into Agda, using the parsed file of the E proof. Regarding to this, there are two main things to be done in the near future: first, we have to decide if working with a new data type defined by us, or not; and we have to start with the development of a sort of library where are included the inference rules from E. It is important to say

[4] `https://github.com/agomezl/tstp2agda`

that it is a cyclic process, because once we start coding we will need to correct mistakes in the parser or adding new inference rules to the library. This means that the library, the parser and the code to perform the proof reconstruction will be in development until the main goal of the project is reached.

References

Bove, A., & Dybjer, P. (2009). Dependent Types at Work. In A. Bove, L. Soares Barbosa, A. Pardo, & J. Sousa Pinto (Eds.), *Language Engineering and Rigourous Software Development* (1st ed., p. 57-99). Springer.

Bove, A., Dybjer, P., & Norell, U. (2009). A Brief Overview of Agda–A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics* (pp. 73–78). Springer.

Da Costa, N., & de Ronde, C. (2014). Non-reflexive Logical Foundation for Quantum Mechanics. *Foundations of Physics*, *44*(12), 1369-1380.

Schulz, S. (2013). System Description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning* (pp. 735–743).

Sicard-Ramírez, A. (2015). *Reasoning about Functional Programs by Combining Interactive and Automatic Proofs* (Unpublished doctoral dissertation). Universidad de la República - Uruguay, Montevideo, Uruguay.

Sutcliffe, G. (2009). The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, *43*(4), 337-362.

Sutcliffe, G., Zimmer, J., & Schulz, S. (2004). TSTP Data-Exchange Formats for Automated Theorem Proving Tools. *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, *112*, 201–215.

Swierstra, S. Doaitse. (2009). Combinator Parsing: A Short Tutorial. In A. Bove, L. Soares Barbosa, A. Pardo, & J. Sousa Pinto (Eds.), *Language Engineering and Rigourous Software Development* (1st ed., p. 252-300). Springer.

Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., & Wischnewski, P. (2009). SPASS Version 3.5. In *Automated Deduction–CADE-22* (pp. 140–145). Springer.