Proof Reconstruction: Parsing Proofs
Project Presentation

Diego Alejandro Montoya-Zapata
dmonto39@eafit.edu.co

Advisor - Andrés Sicard-Ramírez
asr@eafit.edu.co

EAFIT University

June 9th, 2015

Some Definitions I

- **Agda** is a proof assistant. It is an interactive system for writing and checking proofs. Agda is also a functional language with dependent types.[1]

[1] (Bove, Dybjer & Norell, 2009), "A Brief Introduction of Agda – A Functional Language with Dependent Types".
[2] (Bove & Dybjer, 2009), "Dependent Types at Work".
[3] (Sutcliffe, Zimmer & Schulz, 2004), "TSTP Data-Exchange Formats for Automated Theorem Proving Tools".

Some Definitions I

- **Agda** is a proof assistant. It is an interactive system for writing and checking proofs. Agda is also a functional language with dependent types.[1]
- A **dependent type** is a type that depends on elements of other types.[2]

[1] (Bove, Dybjer & Norell, 2009), "A Brief Introduction of Agda – A Functional Language with Dependent Types".
[2] (Bove & Dybjer, 2009), "Dependent Types at Work".
[3] (Sutcliffe, Zimmer & Schulz, 2004), "TSTP Data-Exchange Formats for Automated Theorem Proving Tools".

Some Definitions I

- **Agda** is a proof assistant. It is an interactive system for writing and checking proofs. Agda is also a functional language with dependent types.[1]
- A **dependent type** is a type that depends on elements of other types.[2]
- "**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)."[3]

---

[1] (Bove, Dybjer & Norell, 2009), "A Brief Introduction of Agda – A Functional Language with Dependent Types".

[2] (Bove & Dybjer, 2009), "Dependent Types at Work".

[3] (Sutcliffe, Zimmer & Schulz, 2004), "TSTP Data-Exchange Formats for Automated Theorem Proving Tools".

Some Definitions II

- **TPTP** is a language understood by most of the ATPs.

[4] (Sicard-Ramírez, 2015), "Reasoning about Functional Programs by Combining Interactive and Automatic Proofs".

Some Definitions II

- **TPTP** is a language understood by most of the ATPs.
- **TSTP** is a language for writing the proofs performed by the ATPs.

[4](Sicard-Ramírez, 2015), "Reasoning about Functional Programs by Combining Interactive and Automatic Proofs".

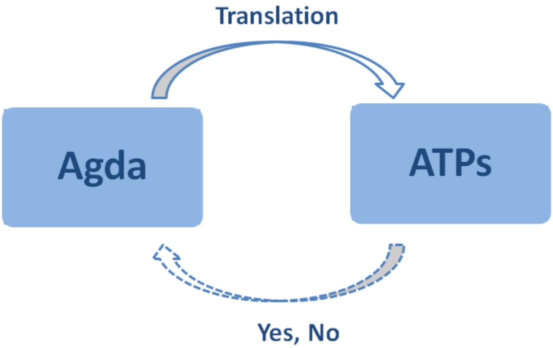
Some Definitions II

- **TPTP** is a language understood by most of the ATPs.
- **TSTP** is a language for writing the proofs performed by the ATPs.
- **Apia** is a program (developed by Prof. Sicard-Ramírez) that performs the translation of an Agda representation of FOL formula into TPTP.[4]

[4] (Sicard-Ramírez, 2015), "Reasoning about Functional Programs by Combining Interactive and Automatic Proofs".

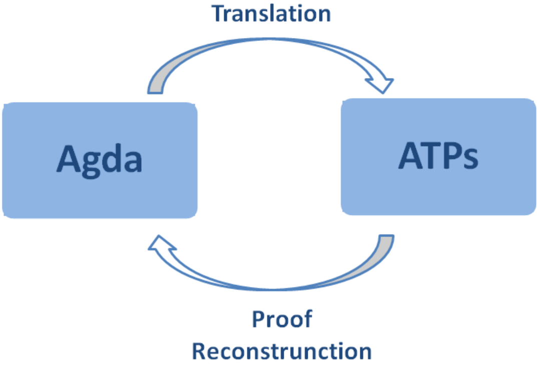
Problem Definition

In this moment:



Problem Definition

What we want:



Goal

The TPTP library has provided the community with standards for input and output for ATPs.[5] 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.

[5](Sutcliffe, 2009), "The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts".

State of the Art

- **SMTCoq** is a Coq tool that checks proof witnesses coming from external SAT and SMT solvers.[6]

[6](Armand, Faure, Grégoire, Keller, Théry & Werner, 2011), "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses".
[7](Blanchette, Bohme, Fleury, Smolka & Steckermeier, 2015), "Semi-intelligible Isar Proofs from Machine Generated Proofs".
[8](Foster & Struth, 2011), "Integrating an Automated Theorem Prover into Agda".

State of the Art

- **SMTCoq** is a Coq tool that checks proof witnesses coming from external SAT and SMT solvers.[6]
- **Sledgehammer** is a component of the Isabelle/HOL proof assistant that integrates external ATPs to discharge interactive proof obligations. Something impressive is that Sledgehammer transforms the proofs by contradiction into direct proofs.[7]

[6] (Armand, Faure, Grégoire, Keller, Théry & Werner, 2011), "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses".

[7] (Blanchette, Bohme, Fleury, Smolka & Steckermeier, 2015), "Semi-intelligible Isar Proofs from Machine Generated Proofs".

[8] (Foster & Struth, 2011), "Integrating an Automated Theorem Prover into Agda".

State of the Art

- **SMTCoq** is a Coq tool that checks proof witnesses coming from external SAT and SMT solvers.[6]
- **Sledgehammer** is a component of the Isabelle/HOL proof assistant that integrates external ATPs to discharge interactive proof obligations. Something impressive is that Sledgehammer transforms the proofs by contradiction into direct proofs.[7]
- Foster and Struth integrated the Waldmeister ATP to Agda.[8]

[6](Armand, Faure, Grégoire, Keller, Théry & Werner, 2011), "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses".

[7](Blanchette, Bohme, Fleury, Smolka & Steckermeier, 2015), "Semi-intelligible Isar Proofs from Machine Generated Proofs".

[8](Foster & Struth, 2011), "Integrating an Automated Theorem Prover into Agda".

Work Done

Before starting with the reconstruction of the proofs, it was necessary to focus on the prerequisites:

- Haskell

Work Done

Before starting with the reconstruction of the proofs, it was necessary to focus on the prerequisites:

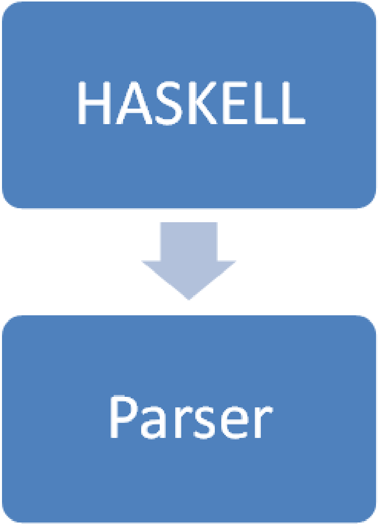
- Haskell
- Agda

Work Done

Before starting with the reconstruction of the proofs, it was necessary to focus on the prerequisites:

- Haskell
- Agda
- The ATP

Haskell



Example[9]

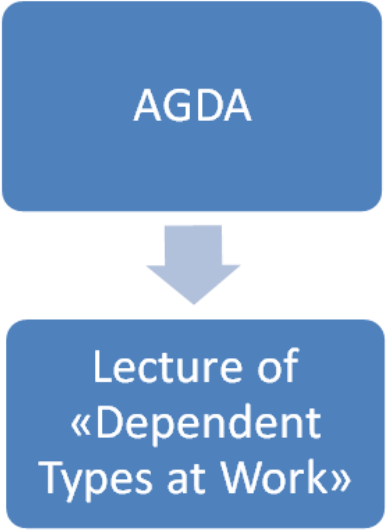
Parsing an IP address in Haskell

```
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 )
```

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

Agda[10]



[10](Bove & Dybjer, 2009), "Dependent Types at Work"

Example[11]

Natural numbers in Agda

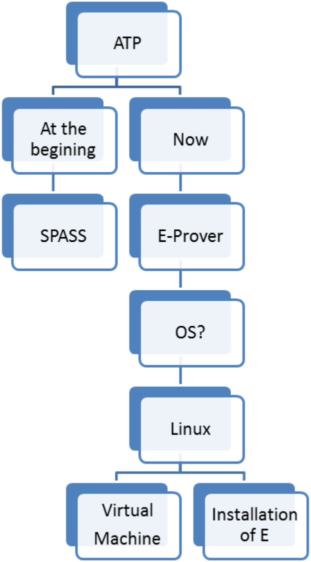
```
data Nat : Set where
  zero : Nat
  succ : Nat -> Nat
```

Equality between Natural numbers in Agda

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

[11] (Bove & Dybjer, 2009), "Dependent Types at Work".

ATP


ATP
At the begining
Now
SPASS
E-Prover
OS?
Linux
Virtual Machine
Installation of E

Example

Proof of the *Modus Ponens* Principle in E

```
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])])).
```

Work in Progress

Currently, we are testing the behavior of the parser from TSTP into Agda, developed by Gómez-Londoño.[12] Its performance is being verified with basic E proofs.

[12] https://github.com/agomezl/tstp2agda

Example

Parsing the Proof of the *Modus Ponens* Principle

```
F {name = "c_0_0", role = Axiom, formula = PredApp (AtomicWord "a") [],
   source = File "modusPonens.tptp" (Just "a")}
F {name = "c_0_3", role = Axiom, formula = PredApp (AtomicWord "a") [],
   source = Name "c_0_0"}
F {name = "c_0_5", role = Axiom, formula = PredApp (AtomicWord "a") [],
   source = Name "c_0_3"}
```

Thanks for your attention!