tstp2agda-0.1.0.0: Proof-term reconstruction from TSTP to Agda

Author
Alejandro Gómez-Londoño

Email
agomezl@eafit.edu.co

Date
December 19 2015

Homepage
https://github.com/agomezl/tstp2agda

Documentation

Issues
https://github.com/agomezl/tstp2agda/issues

Description
A library for translating TSTP proofs into Agda code

To get started see the documentation for T2A module below

To date a number of restrictions and limitations are present

- Only proofs generated by the Metis 2.3 (release 20150303) ATP are supported
- Lack of complete first-order logic support, currently only propositional logic is supported
- Duplication errors with some proofs

Modules

- Data
  - Data.Proof
  - Data.TSTP
- T2A
  - T2A.Core
  - TSTP
  - Util
## Data.Proof

### Types

data **ProofTreeGen a**

Generic tree structure for representing the structure of a proof.

**Constructors**

- **Leaf Role a**
  - `Leaf r a` is a node with `Role r` and content `a` (usually `String`, `F` or `Formula`) and with no dependencies in other nodes.

- **Root Rule a [ProofTreeGen a]**
  - `Root r a d` is a node with deduction `Rule r`, content `a` (usually `String`, `F` or `Formula`), and dependencies `d`.

### Instances

- **Functor ProofTreeGen**
- **Foldable ProofTreeGen**
- **Traversable ProofTreeGen**
- **Eq a => Eq (ProofTreeGen a)**
- **Ord a => Ord (ProofTreeGen a)**
- **Show a => Show (ProofTreeGen a)**

**type** **ProofTree = ProofTreeGen String**

Concrete instance of `ProofTreeGen` with `String` as contents. Each node contains the name of a corresponding formula, along with its dependencies.

**type** **ProofMap = Map String F**

- `Map` from formula names to an `F` formula type, useful to get more information from a node in a `ProofTree`.

**type** **IdSet = Set (Int, String)**

- Simple type for sets of identifiers with associated scopes

### Constructors

**buildProofTree**

- `:: ProofMap -> F` for resolving dependencies
ProofTree

Tree of formulas with the given formula as root

buildProofTree m f, build a ProofTree with f as root, and using m for dependencies resolution. Depending on the root, not all values in m are used.

buildProofMap

:: [F]  List of functions
-> ProofMap  Map of the given functions indexed by its names

buildProofMap lf, given a list of functions lf builds a ProofMap

Internals

getParents

:: ProofMap  Map
-> [Parent]  List of 'Parents
-> [F]  List of parent formulas

getParents m p, from a Map m and a list of parents p returns a list of corresponding parent formulas.

getParentsTree

:: ProofMap  Map
-> [Parent]  List of parents
-> [ProofTree]  List of parents subtrees

getParentsTree m p, from a Map m and a list of parents p return a list of corresponding parent subtrees.

unknownTree

:: Show a
=> String  Description of the unexpected data type
-> a  Unexpected data
-> String  Formula name
-> ProofTree  Unknown node

When an unknown Rule, Source, or other unexpected data type is found a Leaf With an Unknown Role and error message is created.
Data.TSTP

Documentation

data F

Main formula type, it contains all the elements and information of a TSTP formula definition. While name, role, and formula are self-explanatory, source is a messy meta-language in itself, different ATPs may embed different amounts of information in it.

Constructors

F

name :: String
role :: Role
formula :: Formula
source :: Source

Instances

Eq F
Ord F
Read F
Show F

data Role

Formula roles.

Constructors

Axiom
Hypothesis
Definition
Assumption
Lemma
Theorem
Conjecture
NegatedConjecture
Plain
FiDomain
FiFunctors
FiPredicates
Type
# Formulas and terms

**data Formula**

first-order logic formula.

**Constructors**

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BinOp Formula BinOp Formula</td>
<td>Binary connective application</td>
</tr>
<tr>
<td>InfixPred Term InfixPred Term</td>
<td>Infix predicate application</td>
</tr>
<tr>
<td>PredApp AtomicWord [Term]</td>
<td>Predicate application</td>
</tr>
<tr>
<td>Quant Quant [V] Formula</td>
<td>Quantified Formula</td>
</tr>
<tr>
<td>(~:) Formula</td>
<td>Negation</td>
</tr>
</tbody>
</table>

**Instances**

<table>
<thead>
<tr>
<th>Instance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq Formula</td>
<td></td>
</tr>
<tr>
<td>Ord Formula</td>
<td></td>
</tr>
<tr>
<td>Read Formula</td>
<td></td>
</tr>
<tr>
<td>Show Formula</td>
<td></td>
</tr>
<tr>
<td>Show [Formula]</td>
<td></td>
</tr>
</tbody>
</table>

**data Term**

First-order logic terms.

**Constructors**

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var V</td>
<td>Variable</td>
</tr>
<tr>
<td>NumberLitTerm Rational</td>
<td>Number literal</td>
</tr>
<tr>
<td>DistinctObjectTerm String</td>
<td>Double-quoted item</td>
</tr>
<tr>
<td>FunApp AtomicWord [Term]</td>
<td>Function symbol application (constants are encoded as nullary functions)</td>
</tr>
</tbody>
</table>

**Instances**

<table>
<thead>
<tr>
<th>Instance</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq Term</td>
<td></td>
</tr>
<tr>
<td>Ord Term</td>
<td></td>
</tr>
<tr>
<td>Read Term</td>
<td></td>
</tr>
<tr>
<td>Show Term</td>
<td></td>
</tr>
</tbody>
</table>
Show instances

Formula, Term and other data types in this section have Show instances that allow pretty-printing of Formulas and Show [Formula] is an especial instance that print its contents as sequence of implications

```haskell
>>> let f1 = PredApp (AtomicWord "a") []
>>> let f2 = PredApp (AtomicWord "b") []
>>> let f3 = (BinOp (PredApp (AtomicWord "a") []) (:&:) (PredApp (AtomicWord "b") []))
>>> f1 a
>>> f2 b
>>> f3 a \land b
>>> [f1,f2,f3]
{ a b : Set} → a → b → a \land b
```

Some syntax sugar is also present

```haskell
>>> PredApp (AtomicWord "$false") []
⊥
```

class (newtype V)

Variable

Constructors

- `V String`

Instances

- `Eq V`
- `Ord V`
- `Read V`
- `Show V`

data BinOp

Binary formula connectives.

Constructors

- `(:<=:)` ← Equivalence
- `(:=>:)` → Implication
- `(:<:)` ← Reverse Implication
- `(:&:)` \land AND
- `(:|:)` \lor OR
- `(:~&:)` \land \neg NAND
- `(:~|:)` \lor \neg NOR
data **InfixPred**

Infix connectives of the form \( \text{Term} \rightarrow \text{Term} \rightarrow \text{Formula} \).

**Constructors**

\[ (\text{:=}) \quad = \]
\[ (\text{!:=}) \quad \neq \]

**Instances**

- Eq **InfixPred**
- Ord **InfixPred**
- Read **InfixPred**
- Show **InfixPred**

---

**data** **Quant**

Quantifier specification.

**Constructors**

- **All** \( \forall \)
- **Exists** \( \exists \)

**Instances**

- Eq **Quant**
- Ord **Quant**
- Read **Quant**
- Show **Quant**

---

**newtype** **AtomicWord**

**Constructors**

**AtomicWord** \( \text{String} \)

**Instances**

- Eq **AtomicWord**
- Ord **AtomicWord**
- Read **AtomicWord**
Source information

The `Source` data type's main purpose is to provide all the information regarding the deductive process that lead to a given formula. Information about the rules applied along with parent formulas and `SZS` status are among the information you might expect from this field.

Constructors

- `Source String`
- `Inference Rule [Info] [Parent]`
- `Introduced IntroType [Info]`
- `File String (Maybe String)`
- `Theory Theory [Info]`
- `Creator String [Info]`
- `Name String`
- `NoSource`

Instances

- `Eq Source`
- `Ord Source`
- `Read Source`
- `Show Source`

The `Rule` data type represents a deduction rule applied.

Constructors

- `Simplify`
- `Negate`
- `Canonicalize`
- `Strip`
- `NewRule String`

Instances

- `Eq Rule`
- `Ord Rule`
- `Read Rule`
- `Show Rule`

The `Parent` data type is used to store information about the parent of a given rule.
Parent formula information.

Constructors

| Parent String | [GTerm] |

Instances

| Eq Parent |
| Ord Parent |
| Read Parent |
| Show Parent |

Functions

| isBottom :: F -> Bool |
| bottom :: Formula |
| freeVarsF :: Formula -> Set V |
| freeVarsT :: Term -> Set V |
| getFreeVars :: [Formula] -> [V] |

Unused types

The following types are required to have full support of the TSTP syntax but haven't been used yet in tstp2agda aside from the parser.

data IntroType

| Constructor |
| Definition_ |
| AxiomOfChoice |
| Tautology |
| Assumption_ |
data **Theory**

**Constructors**

- **Equality**
- **AC**

**Instances**

- **Eq Theory**
- **Ord Theory**
- **Read Theory**
- **Show Theory**

data **Info**

**Constructors**

- **Description String**
- **IQuote String**
- **Status Status**
- **Function String [GTerm]**
- **InferenceInfo Rule String [GTerm]**
- **AssumptionR [String]**
- **Refutation Source**

**Instances**

- **Eq Info**
- **Ord Info**
- **Read Info**
- **Show Info**

data **Status**

**Constructors**

- **Suc**
- **Unp**
- **Sap**
<table>
<thead>
<tr>
<th>Instances</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Eq Status</td>
<td></td>
</tr>
<tr>
<td>Ord Status</td>
<td></td>
</tr>
<tr>
<td>Read Status</td>
<td></td>
</tr>
<tr>
<td>Show Status</td>
<td></td>
</tr>
</tbody>
</table>

**data GData**

Metadata (the `general_data` rule in TPTP’s grammar)
Constructors

- **GWord** AtomicWord
- **GApp** AtomicWord [GTerm]
- **GVar** V
- **GNumber** Rational
- **GDistinctObject** String
- **GFormulaData** String Formula
- **GFormulaTerm** String Term

Instances

- **Eq** GData
- **Ord** GData
- **Read** GData
- **Show** GData

```
data GTerm

Metadata (the general_term rule in TPTP's grammar)

Constructors

- **ColonSep** GData GTerm
- **GTerm** GData
- **GList** [GTerm]

Instances

- **Eq** GTerm
- **Ord** GTerm
- **Read** GTerm
- **Show** GTerm
```

Produced by Haddock version 2.16.0
How to use tstp2agda

Let use an example, given this problem

```plaintext
$ cat problem.tstp
fof(a1,axiom,a).
fof(a2,conjecture,a).
```

and the corresponding Metis proof

```plaintext
$ cat proof.tstp
-----------------------------------------------------------------------------------
S Z S  status Theorem for examples problemTest-1.tstp
S Z S  output start CNFRefutation for examples problemTest-1.tstp
fof(a1, axiom, (a)).
fof(a2, conjecture, (a)).
fof(subgoal_0, plain, (a), inference(strip, [], [a2])).
fof(negate_0_0, plain, (~ a), inference(negate, [], [subgoal_0])).
fof(normalize_0_0, plain, (~ a),
    inference(canonialize, [], [negate_0_0])).
fof(normalize_0_1, plain, (a), inference(canonialize, [], [a1])).
fof(normalize_0_2, plain, ($false),
    inference(simplify, [], [normalize_0_0, normalize_0_1])).
cnf(refute_0_0, plain, ($false),
    inference(canonialize, [], [normalize_0_2])).
S Z S  output end CNFRefutation for examples problemTest-1.tstp
```

we create some required data structures

```plaintext
main :: IO ()
main = do
  -- read the file
  formulas <- parseFile "proof.tstp"
  -- create a map
  proofmap <- buildProofMap formulas
  -- get subgoals, refutes, axioms, and the conjecture
  let subgoals = getSubGoals formulas
  let refutes = getRefutes formulas
  let axioms = getAxioms formulas
  let (Just conj) = getConjecture formulas
  -- build a proof tree for each subgoal (using his associated refute)
  let prooftree = map (buildProofTree proofmap) refutes
```

and then print the actual Agda code

```plaintext
-- PREAMBLE: module definitions and imports
printPreamble "BaseProof"
-- STEP 1: Print auxiliary functions
printAuxSignatures proofmap prooftree
-- STEP 2: Subgoal handling
printSubGoals subgoals conj "goals"
-- STEP 3: Print main function signature
printProofBody axioms conj "proof" subgoals "goals"
-- STEP 4: Print all the step of the proof as local definitions
mapM_ (printProofWhere proofmap prooftree)
printSubGoals subgoals conj "goals"
-- STEP 3: Print main function signature
printProofBody axioms conj "proof" subgoals "goals"
-- STEP 4: Print all the step of the proof as local definitions
mapM_ (printProofWhere proofmap prooftree)

and then get

module BaseProof where
  open import Data.FOL.Shallow
  postulate fun-normalize-0-0 : { a : Set} → ¬ a → ¬ a
  postulate fun-normalize-0-1 : { a : Set} → a → a
  postulate fun-normalize-0-2 : { a : Set} → ¬ a → a → ⊥
  postulate fun-refute-0-0 : ⊥ → ⊥
  postulate goals : { a : Set} → a → a
  proof : { a : Set} → a → a
  proof {a} a1 = goals subgoal-0
  where
    fun-negate-0-0 : ¬ a → ⊥
    fun-negate-0-0 negate-0-0 = refute-0-0
    where
      normalize-0-0 = fun-normalize-0-0 negate-0-0
      normalize-0-1 = fun-normalize-0-1 a1
      normalize-0-2 = fun-normalize-0-2 normalize-0-0 normalize-0-1
      refute-0-0 = fun-refute-0-0 normalize-0-2
      subgoal-0 = proofByContradiction fun-negate-0-0

Getters

getSubGoals :: [F] -> [F]

Extract subgoals from a list of formulae.

getRefutes :: [F] -> [F]

Extract refuting steps from a list of formulae.

getAxioms :: [F] -> [F]

Extract axioms from a list of formulae.

getConjecture :: [F] -> Maybe F

Try to extract a conjecture from a list of formulae and checks for uniqueness.

Agda translation

printPreamble :: String

Module name
printAuxSignatures

:: ProofMap : map of formulas
-> [ProofTree] : list of subgoals
-> IO ()

Print a series of auxiliary functions required to perform most of the steps of the proof. Every Formula has a corresponding function which has its parents as arguments and the current function as return value. Since a proof is split in various subgoals, this function receives a list of sub-ProofTrees.

fun-step_m_n : { ∀ i : Set } → step_m_n1 → ... → step_m_nj → step_m_n

printSubGoals

:: [F] : Subgoals
-> F : Conjecture
-> String : Function name (subGoalImplName)
-> IO ()

Print the main subgoal implication function

subGoalImplName : subgoal_0 → ... → subgoal_n → conjecture

printProofBody

:: [F] : Axioms
-> F : Conjecture
-> String : Function name (proofName)
-> [F] : Subgoals
-> String : Name of subGoalImplName
-> IO ()

Print main proof type signature, and top level LHS ans RHS of the form

proofName : axiom_0 → ... → axiom_n → conjecture
proofName axiomName_0 ... axiomName_n = subGoalImplName subgoal_0 ... subgoal_n where

printProofWhere :: ProofMap -> ProofTree -> IO ()

For a given subgoal print each formula definition in reverse order of dependencies

negation_0 : ¬ τ_0 → ⊥
negation_0 negate_0 = refute_0
where
step_0_i = fun-step_0_i step_0_i1 ... step_0_ij
This is perhaps the most important step and the one that does the "actual" proof translation. The basic principle is to define each subgoal in terms of its parents which for most (if not all) cases implies a negation of the subgoal and a corresponding refute term.

### buildProofMap

<table>
<thead>
<tr>
<th align="left">:: [F]</th>
<th align="left">List of functions</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left">-&gt; ProofMap</td>
<td align="left">Map of the given functions indexed by its names</td>
</tr>
</tbody>
</table>

**buildProofMap** \( \text{lf} \), given a list of functions \( \text{lf} \) builds a **ProofMap**

### buildProofTree

<table>
<thead>
<tr>
<th align="left">:: ProofMap</th>
<th align="left">Map for resolving dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td align="left">-&gt; F</td>
<td align="left">Root formula</td>
</tr>
<tr>
<td align="left">-&gt; ProofTree</td>
<td align="left">Tree of formulas with the given formula as root</td>
</tr>
</tbody>
</table>

**buildProofTree** \( \text{m} \), \( \text{f} \), build a **ProofTree** with \( \text{f} \) as root, and using \( \text{m} \) for dependencies resolution. Depending on the root, not all values in \( \text{m} \) are used.

### TSTP parsing

| parseFile :: Maybe FilePath -> IO [F] |
| :-- | :-- |

Similar to **parse** but reading directly from a file or stdin.
## T2A.Core

### Documentation

data AgdaSignature

An Agda type signature $\alpha : \tau$

#### Constructors

- **Signature String [Formula]**  
  Regular top level signature

- **ScopedSignature String [Formula]**  
  Fully scoped signature with no newly introduced type variables

#### Instances

- **Eq AgdaSignature**
- **Ord AgdaSignature**
- **Show AgdaSignature**

**buildSignature :: ProofMap -> String -> Maybe AgdaSignature**

Given a proof map $\omega$ and some formula name $\varphi$, construct the appropriated AgdaSignature based on the parents of $\varphi$

**fname :: AgdaSignature -> String**

Retrieve signature name
TSTP

Documentation

parse :: String -> [F]

Parse a TSTP file and return a list of F formulas in no particular order, for example:

\[
\text{\$ cat examples/proof/Basic-1.tstp}
\]
\[
\text{fof(a1, axiom, (a)).} \\
\text{fof(a2, axiom, (b)).} \\
\text{fof(a3, axiom, ((a & b) => z)).} \\
\text{...}
\]

would be:

\[
[ \\
  F \{ \text{name = "a1", role = Axiom, formula = a, source = NoSource}\}, \\
  F \{ \text{name = "a2", role = Axiom, formula = b, source = NoSource}\}, \\
  F \{ \text{name = "a3", role = Axiom, formula = ( a \land b \rightarrow z ), source = NoSource}\}, \\
  ...
\]
\]

parseFile :: Maybe FilePath -> IO [F]

Similar to parse but reading directly from a file or stdin.
## Spaced concatenation

\[
\text{forall } a, b. (\text{BShow } a, \text{BShow } b) \Rightarrow a \to b \to \text{String}
\]

infixr 4

"foo" \* "bar" = "foo bar". Its main use is to simplify the concatenation of printable types separated by spaces.

```haskell
class BShow a where
  BShow fixes Show String instance behavior length "foo" \neq length (show "foo") with two new instances (and overlapped) instances for String and Show a.

Methods

\[ \text{\texttt{\text{\texttt{betaShow}}}} :: a \to \text{String} \]

Instances

BShow Char
BShow String
Show a \Rightarrow BShow a
```

## Printing with indentation

```haskell
printInd :: Show a \Rightarrow Int \to a \to IO ()
printInd i b, prints a with b level of indentation i.
```

```haskell
putStrLnInd :: Int \to String \to IO ()
printInd i str, prints a with str level of indentation i.
```

## List manipulation

```haskell
unique :: Ord a \Rightarrow [a] \to [a]
```

Removes duplicate elements of a list.

```haskell
swapPrefix :: Eq a
  => [a] \to [a] \to [a] \to [a] \to [a]
```

Current prefix

Replacement

List to be replaced

Resulting list
swapprefix a b str, replaces prefix a in str with b checking that a is a prefix of str.

Others

<table>
<thead>
<tr>
<th>agdafy :: String -&gt; String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metis identifiers usually contain _ characters which are invalid in Agda, agdafy replaces normalize_0_0 with normalize-0-0. This is mostly used inside the Happy parser every time an AtomicWord is created.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stdout2file :: Maybe FilePath -&gt; IO ()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redirect all stdout output into a file or do nothing (in case of Nothing)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>checkIdScope :: Int -&gt; String -&gt; Set (Int, String) -&gt; Bool</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkIdScope i t s check if any name in s has a more general scope than t with level i</td>
</tr>
</tbody>
</table>

Produced by Haddock version 2.16.0