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
http://agomezl.github.io/doc/html/tstp2agda/index.html
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.Proof

Safe Haskell	None
Language	Haskell2010

Synopsis

Contents

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 whit associated scopes

Constructors

```
buildProofTree
```

`:: ProofMap`	`Map` for resolving dependencies
`-> F`	Root formula

`-> 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

Safe Haskell	None
Language	Haskell2010

Synopsis

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
```

Unknown

Instances

```
Eq Role
Ord Role
Read Role
Show Role
```

Formulas and terms

data **Formula**

first-order logic formula.

Constructors

BinOp `Formula BinOp Formula`	Binary connective application
InfixPred `Term InfixPred Term`	Infix predicate application
PredApp `AtomicWord [Term]`	Predicate application
Quant `Quant [V] Formula`	Quantified Formula
(:~:) `Formula`	Negation

Instances

```
Eq Formula
Ord Formula
Read Formula
Show Formula
Show [Formula]
```

data **Term**

First-order logic terms.

Constructors

Var `V`	Variable
NumberLitTerm `Rational`	Number literal
DistinctObjectTerm `String`	Double-quoted item
FunApp `AtomicWord [Term]`	Function symbol application (constants are encoded as nullary functions)

Instances

```
Eq Term
Ord Term
Read Term
Show Term
```

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

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

Some syntax sugar is also present

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

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*
`(:&:)`	∧ *AND*
`(:|:)`	∨ *OR*
`(:~&:)`	$\overline{\wedge}$ *NAND*
`(:~|:)`	$\overline{\vee}$ *NOR*

(:<~>:) ⊕ *XOR*

Instances

```
Eq BinOp
Ord BinOp
Read BinOp
Show BinOp
```

data **InfixPred**

Infix connectives of the form *Term -> Term -> Formula*.

Constructors

(:=:)	=
(:!=:)	≠

Instances

```
Eq InfixPred
Ord InfixPred
Read InfixPred
Show InfixPred
```

data **Quant**

Quantifier specification.

Constructors

All	∀
Exists	∃

Instances

```
Eq Quant
Ord Quant
Read Quant
Show Quant
```

newtype **AtomicWord**

Constructors

```
AtomicWord String
```

Instances

```
Eq AtomicWord
Ord AtomicWord
Read AtomicWord
```

Show AtomicWord

Source information

data **Source**

Source 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

data **Rule**

Deduction rule applied.

Constructors

Simplify
Negate
Canonicalize
Strip
NewRule String

Instances

Eq Rule
Ord Rule
Read Rule
Show Rule

data **Parent**

Parent formula information.

Constructors

```
Parent String [GTerm]
```

Instances

```
Eq Parent
Ord Parent
Read Parent
Show Parent
```

Functions

```
isBottom :: F -> Bool
```

`isBottom f`, test whether `f` = ⊥.

```
bottom :: Formula
```

`bottom` = ⊥.

```
freeVarsF :: Formula -> Set V
```

`freeVarsF f`, returns a `Set` of all free variables of `f`.

```
freeVarsT :: Term -> Set V
```

`freeVarsT t`, returns a `Set` of all free variables of `t`.

```
getFreeVars :: [Formula] -> [V]
```

`getFreeVars f`, given a list of formulas `f` return all free variables in the formulas.

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
```

Constructors

```
Definition_
AxiomOfChoice
Tautology
Assumption_
```

UnknownType

Instances

Eq IntroType
Ord IntroType
Read IntroType
Show IntroType

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

Esa
Sat
Fsa
Thm
Eqv
Tac
Wec
Eth
Tau
Wtc
Wth
Cax
Sca
Tca
Wca
Cup
Csp
Ecs
Csa
Cth
Ceq
Unc
Wcc
Ect
Fun
Uns
Wuc
Wct
Scc
Uca
Noc
Unk

⊟ **Instances**

`Eq Status`
`Ord Status`
`Read Status`
`Show Status`

`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
```

T2A

Safe Haskell None
Language Haskell2010

Synopsis

Contents

How to use tstp2agda

Let use an example, given this problem

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

and the corresponding Metis proof

```
$ cat proof.tstp
--------------------------------------------------------------------------------
SZS status Theorem for examplesproblemTest-1.tstp
SZS output start CNFRefutation for examplesproblemTest-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(canonicalize, [], [negate_0_0])).
fof(normalize_0_1, plain, (a), inference(canonicalize, [], [a1])).
fof(normalize_0_2, plain, ($false),
    inference(simplify, [], [normalize_0_0, normalize_0_1])).
cnf(refute_0_0, plain, ($false),
    inference(canonicalize, [], [normalize_0_2])).
SZS output end CNFRefutation for examplesproblemTest-1.tstp
```

we create some requiered data structures

```
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) = getConjeture formulas
  -- build a proofTree for each subgoal (using his associated refute)
  let prooftree = map (buildProofTree proofmap) refutes
```

and then print the actual Agda code

```
  -- 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.

```
getConjeture :: [F] -> Maybe F
```

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

Agda translation

```
printPreamble
```

`:: String` Module name

```
-> IO ()
```

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-`ProofTree`s

```
fun-step_m_n : { ν₀ ... νᵢ : 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 : subgoal₀ → ... → subgoalₙ → 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 : axiom₀ → ... → axiomₙ → conjecture
proofName axiomName₀ ... axiomNameₙ = subGoalImplName subgoal₀ ... subgoalᵣ
  where
```

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

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

```
    negation₀ : ¬ τ₀ → ⊥
    negation₀ negate₀ = refute₀
      where
        step₀_i = fun-step₀_i step₀_i1 .. step₀_ij
```

```
        ...
        step₀_₀ = fun-step₀_₀ step₀_₀₁ .. step₀_₀ₖ
    subgoal₀ = proofByContradiction negation₀
    ...
    negationₙ : ¬ τₙ → ⊥
    negationₙ negateₙ = refuteₙ
      where
        stepₙ_ₜ = fun-stepₙ_ₜ stepₙ__ₜ₁ .. stepₙ_ₜₒ
        ...
        stepₙ_₀ = fun-stepₙ_₀ stepₙ_₀₁ .. stepₙ_₀ᵤ
    subgoal₀ = proofByContradiction negationₙ
```

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

`:: [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`

buildProofTree

`:: ProofMap`	`Map` for resolving dependencies
`-> F`	Root formula
`-> 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.

TSTP parsing

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

Similar to `parse` but reading directly from a file or stdin.

T2A.Core

Safe Haskell	None
Language	Haskell2010

Synopsis

Documentation

data AgdaSignature

An Agda type signature α : τ

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 ω and some formula name φ, construct the appropriated AgdaSignature based on the parents of φ

fname :: AgdaSignature -> String

Retrieve signature name

TSTP

Safe Haskell	None
Language	Haskell2010

Documentation

Synopsis

```
parse :: String -> [F]
```

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

```
$ cat examples/proof/Basic-1.tstp
fof(a1, axiom, (a)).
fof(a2, axiom, (b)).
fof(a3, axiom, ((a & b) => z)).
...
```

would be:

```
[
  F {name = "a1", role = Axiom, formula = a, source = NoSource},
  F {name = "a2", role = Axiom, formula = b, source = NoSource},
  F {name = "a3", role = Axiom, formula = ( a ∧ b → z ), source = NoSource},
  ...
]
```

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

Similar to `parse` but reading directly from a file or stdin.

Util

Safe Haskell	None
Language	Haskell2010

Synopsis

Spaced concatenation

```
(▪) :: forall a b. (BShow a, BShow b) => a -> b -> String
  infixr 4
```

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

```
class BShow a where
```

`BShow` fixes `Show String` instance behavior `length "foo" ≠ length (show "foo")` with two new instances (and overlapped) instances for `String` and `Show a`.

Methods

```
βshow :: a -> String
```

Instances

- `BShow Char`
- `BShow String`
- `Show a => BShow a`

Printing with indentation

```
printInd :: Show a => Int -> a -> IO ()
```

`printInd i b`, prints a with b level of indentation `i`.

```
putStrLnInd :: Int -> String -> IO ()
```

`printInd i str`, prints a with `str` level of indentation `i`.

List manipulation

```
unique :: Ord a => [a] -> [a]
```

Removes duplicate elements of a list.

```
swapPrefix
```

`:: Eq a`	
`=> [a]`	Current prefix
`-> [a]`	Replacement
`-> [a]`	List to be replaced
`-> [a]`	Resulting list

`swapPrefix a b str`, replaces prefix `a` in `str` with `b` checking that `a` is a prefix of `str`.

Others

`agdafy :: String -> String`

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.

`stdout2file :: Maybe FilePath -> IO ()`

Redirect all stdout output into a file or do nothing (in case of `Nothing`)

`checkIdScope :: Int -> String -> Set (Int, String) -> Bool`

`checkIdScope i t s` check if any name in `s` has a more general scope than `t` with level `i`