Reference Manual

Title: Proof Checker Reference Manual
Author: Yu-Yang Lin
Date: 22 August, 2015

Proof Checker Reference Manual

`version: 0.9.0.1`

Table of Contents

Introduction

The Proof Checker is a command-line tool made to validate simple program-correctness proofs of functional programs using inductive and equational reasoning. By necessity, the checker was also made to deal with propositional logic.

The tool source is written in **OCaml**, using OCamllex and Menhir for the parser, and was developed on a Windows machine. Proofs are validated through sequent calculus rules. Specifications for the rules can be seen in the notes.

Usage

Compiling

To use the tool, it must first be compiled. For this, use `src\Makefile` provided. The compiled tool will be named `proof_checker.exe` by default.

Requirements for compilation are:

- GNU Make
- Menhir
- OCamllex
- OCaml Batch Compiler (`ocamlc`)

Compiling on Windows Machines

Windows users can find the Menhir package `godi-menhir` in WODI, the Windows version of the package manager GODI. I recommed this over OPAM since I couldn't get OPAM working.

You can compile the tool from Cygwin. I used make for Windows found in the GnuWin32 files page, and compile the tool from **Windows PowerShell**.

Using the tool

To use the tool, run `proof_checker.exe` from a command-line console on a target proof file. By default, the extension for proof files is `.proof`.

e.g.

```
.\proof_checker.exe "testing\test_proof.proof"
```

This should output a success message if the proof is valid, or output an error message with position data for where the checker failed.

For instance, a successful proof would output:

```
***Opening file: ..\extra\sample_proofs\rev_involution.proof.....[done]***
 ***Lexing and Parsing file.........[done]***
 ***Checking file...................[done]***
 ***VALIDATION SUCCESSFUL****
```

While a failed proof might output:

```
***Opening file: ..\extra\sample_proofs\rev_involution.proof.....[done]***
 ***Lexing and Parsing file.........[done]***
 ***Checking file...................[error]***
 [VALIDATION FAILURE]:
Expected hypothesis for 't=t:(nat) list' but
'[rev xs]' points to proposition
'(forall xs : (nat) list . (forall x : nat . ((rev x :: xs) =
 ((append (rev xs)) x :: nil) : (nat) list)))'.
Encountered while evaluating 'by equality' clause.
Encountered while evaluating 'by induction on list'.
  (line 106 , col 12) to (line 106 , col 34)
```

Syntax for proof files can be seen in the sample proofs provided, you can find the proofs under extra/sample_proofs.

The Proof Language

The Proof Checker uses a language designed to look like a hand-written proof in English. It won't always be grammatically correct English, however.

Keywords and Lexical Conventions

Proofs contain several data type categories which separate the proof files into different hierarchical layers. Organisation of proof files will be explained later on.

Types

This data type contains the type of terms.

- **Booleans**: `bool`
- **Natural Numbers**: `nat`
- **Type Variables**: string starting with apostrophe (`'`) . e.g.`'a`
- **Lists**: `a list` where `a` is a type. e.g. `nat list`
- **Functions**: `a -> b` where `a` and `b` are types. e.g. `nat -> nat`
- **Proposition Type**: `prop`

Terms

This data type category contains terms which give values to the different existing types.

- **Term Variables**:
  - strings starting with lower-case letter, allowing `0-9` and `_`. e.g. `term_var_x`
- **Function Application**:
  - term applied to another term. e.g. `reverse xs`, where `reverse` is a term variable of type `nat list-> nat list` and `xs` is a term variable of type `nat list`.
- **Boolean Terms**:
  - `true` and `false` of type `bool`
- **Natural Numbers**:
  - `zero` and `suc n` where `n` is a term of type `nat`. e.g. `suc (suc zero)`.
- **Lists**:
  - `nil` or `[]` and `x :: xs` where `x` is a term of some type `a`, and `xs` is a list of the same type (`a list`).

Propositions

This data type category contains propositions, which are the type for proofs and labelled by hypotheses. Propositions can contain terms in them, which is how terms are checked in the hierarchy.

- **Truth** and **Falsity**: `Truth` and `Falsity` respectively
- **Propositional Variables**:
  - strings starting with upper-case letter, allowing `0-9` and `_`. e.g. `PROP_VAR_A`.
- **Conjunction**:
  - `A and B` where `A` and `B` are propositions.
- **Disjunction**:
  - `A or B` where `A` and `B` are propositions.
- **Implication**:
  - `A => B` where `A` and `B` are propositions.
- **Equality**:

  - `t_1 = t_2 : type` where `t_1` and `t_2` are terms and `type` is a type shared by both `t_1` and `t_2`. e.g. `suc n = suc n : nat`.

- **Universal and Existential Quantifiers**:
  - `forall x : type . A` where `x` is a term variable, `type` is a type, and `A` is a proposition. e.g. `forall n : nat . suc n = suc n : nat`.
  - `exists x : type . A` where `x` is a term variable, `type` is a type, and `A` is a proposition. e.g. `exists x : bool . x = true : bool`.

Proofs

The proof data type is made up of a set of rules that allow us to prove propositions. Proofs allow us to manipulate propositions, terms, and types in order to show a theorem holds.

- **Truth Introduction**: `tt` is the proof for a `Truth` proposition
- **Falsity Elimination**: `by absurdity of [H]` where `[H]` is a hypothesis
- **Conjunction Introduction**: `(p , q)` where `p` and `q` are proofs.
- **Conjunction Elimination**:

  ```
  we know ([P] : P , [Q] : Q) because [P and Q] : P and Q . rest
  ```

  where `[P]` and `[Q]` are hypotheses of type `P` and `Q`, which are propositions, and `rest` is a proof where `[P]` and `[Q]` are in scope.

- **Disjunction Introduction**:
  - `p on the left` is a proof for `P or Q` where `p` is of type `P` and `Q` is any other proposition.
  - `q on the right` is a proof for `P or Q` where `q` is of type `Q` and `P` is any other proposition.

- **Disjunction Elimination**:

  ```
  since [A or B] : A or B then either :
  case on the left : [A] : A . p
  case on the right : [B] : B . q
  ```

  where `[A or B]` is a hypothesis of type `A or B`, `[A]` is an hypothesis expected to be of type `A`, `[B]` is a hypothesis expected to be of type `B`, `p` is a proof where `[A]` is in scope, and `q` is a proof where `[B]` is in scope.

  When using this proof rule, `left` and `right` must be eliminated in that order. Thus, `case on the left` must be written before `case on the right`.

- **Implication Introduction**:
  - `assume [A] : A . p` where `[A]` is a hypothesis of type `A`, `A` is a proposition, and `p` is a proof where `[A] : A` is in scope.
- **Implication Elimination**:

  ```
  we know [B] : B because [A to B] : A => B with ([A]) . rest
  ```

  where `[B]` is a new hypothesis of type `B`, `[A to B]` is an existing hypothesis of type `A => B`, `[A]` an existing hypothesis of type `A`, and `rest` a proof where the new hypothesis `[B]` is in scope.

  Note that this rule is actually the combination of two rules, a hypothesis labelling clause (`we know [H] because p`), and a `with` clause (`[H] with (a,b,c)`). This will be mentioned in more detail in their own sections.
- **Existential Introduction**:
  - `choose t . rest` where `t` is a term and `rest` is a proof where `t` is now replacing the variable.
- **Existential Elimination**:

  ```
  we know [new A] : A with x because [A] : exists x : type . A . rest
  ```

  where `[new A]` is a new hypothesis where the existential surrounding `A` has been eliminated, `x` is term variable of type `type`, `[A]` is an old hypothesis of type `A`, `A` is a proposition, and `rest` is a proof where `[new A]` and `x` are in scope.
- **Universal Introduction**:
  - `assume x : type . rest` where `x` is a term variable of type `type`, and `rest` is a proof where `x` is now in scope.
- **Universal Elimination**:

  ```
  we know [y A] : A because [A] : forall x : type . A with (y) . rest
  ```

  where `[y A]` is a hypothesis of type `A` where all instances of `x` have been replaced with term `y` in `A`, `[A]` a hypothesis, and `rest` is a proof where `[y A]` is in scope.

  Note that this rule is actually the combination of two rules, a hypothesis labelling clause (`we know [H] because p`), and a `with` clause (`[H] with (a,b,c)`). This will be mentioned in more detail in their own sections.
- **Induction on Natural Numbers**:

```
by induction on nat :
case zero : p
case (suc n) : [IH] : A . q
```

where in `case zero`, `p` is a proof of the proposition where the variable we are applying induction to is replaced with `zero`; and in `case (suc n)`, `suc n` is the `nat` replacing the variable in the inductive step, `[IH]` is the inductive hypothesis where the variable is replaced with `n`, and `q` is a proof where `n` and `[IH]` are in scope.

- **Induction on Lists**:

```
by induction on list :
case [] : p
case (x :: xs) : [IH] : A . q
```

where in `case []`, `p` is a proof of the proposition where the variable we are applying induction to is replaced with `[]`; and in `case (x :: xs)`, `(x :: xs)` is the `list` replacing the variable in the inductive step, `[IH]` is the inductive hypothesis where the variable is replaced with `xs`, and `q` is a proof where `x`, `xs` and `[IH]` are in scope.

- **Induction on Booleans**:

```
by induction on bool :
case true : p
case false : q
```

where `p` is a proof for the proposition with the inductive variable replaced with `true`, and `q` is a proof for the proposition with the inductive variable replced with `false`.

- **Equality**:
  - `equality on ([H_1],[H_2],...,[H_n])` where `[H_1]` to `[H_n]` are the hypotheses used to prove equality of the desired terms, which would be stated by the proof's goal or statement.

- **Hypothesis Labelling Clause**:

```
we know [A] : A because p . rest
```

where `[A]` is a proof of type `A`, `p` is a proof for `A`, and `rest` is a proof where `[A]` is in scope. This is a form of hypothesis introduction, it labels a proven proposition with a hypothesis, and put's it in scope of the following proof.

Given this rule is useful to keep moving forward in a proof, and use proven propositions later in the same proof, it's commonly paired with almost every rule in the proof data type.

Note that you cannot give any proof after a `because` keyword. You can only give what is called a `simple proof`, `tt`, `(p , q)`, `p on left`, `q on right`, `equality on ([A],[B],[C])`, `by [H]`, `[H] with (a,[A])`, and `p therefore A`. i.e. no case elimination rules such as induction or disjunction elimination.

- **With Clause**:

  ```
  [H] with (a,b,c,[A],[B],C])
  ```

  where `[H]` is a hypothesis labelling some proposition made of universal quantifiers or implications, and `a` to `c` are terms, while `[A]` to `[C]` are hypotheses.

  The `with` clause is a form of elimination clause for universal quantifiers and implications. To eliminate a universal, provide a term that can replace the universal term variable. To eliminate an implication, provide a hypothesis which matches the proposition being eliminated.

  Note that the elements we are providing for the elimination must be given inside a tuple, and in the correct order. i.e. When eliminating, you can only eliminate the outermost layer first.

  For instance, to eliminate the following:

  ```
  [Some Hypothesis] : A => forall x : nat . C
  ```

  We must first eliminate `A`, and then `x`.

  Given: `[A] : A` and `y : nat`, we can do:

  ```
  [Some Hypothesis] with ([A],y)
  ```

- **Using Hypotheses**:

  – `by [H]` where `[H]` is the hypothesis we want to use. Note that you must also include the `by` when combining this with a labelling clause. e.g.

    ```
    we know [A] : A because by [H] . p
    ```

- **Therefore Clause**:

  – `p therefore A` where `p` is a proof of type `A`. This is mainly to label your proof with the proposition if it's confusing.
    e.g. Given `[negation] : Falsity` we can do: `by [negation] therefore Falsity`

Top-Level

The top-level data type category contains the outermost hierarchical layer of the proof files. The first thing you write in a proof file is a top-level construct. In these, you can write proofs, propositions, and terms. A proof file can have multiple occurrences (or none) of these top-level constructs.

- **Signatures**:

  ```
  Signatures:
      A : prop ;
      B : prop ;
      append : nat list -> nat list -> nat list ;
      rev    : nat list -> nat list ;
  ```

  where the word `Signatures:` is followed by a list of `variables` with corresponding `types`, each separated by a semi-colon (`;`).

  Signatures are what contain the initial term and proposition variables to feed into the rest of the file's context. Every variable that appears in a `Signature` will be globally available (in scope) of every top-level construct underneath it.

  This is because adding a variable into a `Signature` is the same as adding that variable into the context of the proof file at that point. Another way to think of `Signatures` is to think of them as appending to the `variables context` for the proof file at a given point.

  The reason why I refer to context variables as `Signatures` is because variables with corresponding types are analogous to signatures in functional programs. This is why we can define a `function signature` such as `rev : nat list -> nat list` and `append : nat list -> nat list -> nat list` in this section.

- **Definitions**:

  ```
  Definitions:
      [A to B] : A => B ;
      [not A]  : (A => Falsity) ;
      [DNE]    : ((A => Falsity) => Falsity) => A ;
     [append nil] : forall xs : nat list . append [] xs = xs : nat list ;
      [append xs]  : forall xs : nat list .
                          forall x : nat .
                              forall ys : nat list .
                                  append (x::xs) ys =
                                  x :: append xs ys : nat list ;
      [rev nil] : rev [] = [] : nat list ;
      [rev xs]  : forall xs : nat list .
  ```

```
forall x : nat .
    rev (x :: xs) =
     append (rev xs) (x :: []) : nat list ;
```

where the word `Definitions` is followed by a list of `hypotheses` with corresponding `proposition`, each separated by a semi-colon (`;`).

`Definitions` are the `axioms` or `hypothesis context` of a proof file. It contains propositions that simply hold true, i.e. the `premises` of a `Theorem`. Note that you can define propositions that cannot be proven in this section, such as `double negation elimination`, which is necessary for classical logic.

All given `Definitions` will be fed into the rest of the file's `hypothesis context`, and thus be globally available (in scope) of any top-level construct underneath it.

`Definitions` get their name from the analogy to `program definitions` in functional programs. For instance, in `Haskell`, after writing a `function signature`, we provide a `function definition` that states what the program of the function is, i.e. what the function does.

- **Theorems**:

```
Theorem [P to Z]:
    Statement: P => Z
    Proof:
        assume [P] : P .
        we know [Q] : Q because [P to Q] with ([P]) .
        we know [Z] : Z because [Q to Z] with ([Q]) .
        by [Z]
    QED.
```

where the word `Theorem` is followed by a hypothesis to call the theorem - in this case `[P to Z]`. The keyword `Statement` is followed by the proposition that we are trying to prove in this theorem labelled `[P to Z]`, which is `P => Z`. The keyword `Proof` is followed by the proof for the `Statement` of this `Theorem`.

`Theorems` are like `Definitions` in the sense that they add `hypotheses` into the `hypothesis context` of a proof file. Every proven `Theorem` will be available globablly (in scope) to any top-level constructs underneath it.

`Theorems` allow us to to prove propositions. This is where the `Proof Checker` does its main job, which is to validate the correctness of a given proof. `propositions`, `terms` and `types` will also be checked for well-formedness when fed into the file as a `Signature` or `Definition`, but proofs can only be checked within a `Theorem`.

All Theorem proofs must end with the keyword `QED.`.

Organisation of Proof Files

Under the hood, a proof file contains two data-structures which the proof checker has to keep track of:

- **Variable context**:
  - This is the set where all `term` and `proposition` variables are held. This is a set of pairs `variable , type`, so the checker can tell what type any given variable has.
- **Hypothesis context**:
  - This is the set where all `hypotheses` are held. This is a set of pairs `hypothesis , proposition`, so the checker can tell what proposition any given hypothesis is labelling.

The `top-level` constructs simply feed into either of these contexts if they pass their corresponding check:

- **Signatures**:
  - These add to the variable context only if the `type` is well-formed. For instance:

    ```
    Signatures:
        A   : prop ;
        rev : nat list -> nat list ;
    ```

    will only add `A` and `rev` into the variable context if `prop` and `nat list -> nat list` are well-formed `types`.
- **Definitions**:
  - These add to the hypothesis context only if the `proposition` is well-formed. For instance:

    ```
    Definitions:
        [A to B] : A => B ;
        [not A]  : (A => Falsity) ;
        [plus 1] : forall n : nat . suc n
    ```

    will only add `[A to B]`, `[not A]` and `[plus 1]` into the hypothesis context if `A => B`, `(A => Falsity)`, and `forall n : nat . suc n` are respectively well-formed `propositions`.
- **Theorems**:

– These add to the hypothesis context the `Theorem` label, and its corresponding `Statement` only if the `Statement` is a well-formed proposition and the proof provided under `Proof` is valid. For instance:

```
Theorem [not (P and not P)]:
    Statement: (P and (P => Falsity)) => Falsity
    Proof:
        assume [P and not P] : P and (P => Falsity) .
        we know [P] : P , [not P] : P => Falsity
            because [P and not P] .
        we know [negation] : Falsity
            because [not P] with ([P]) .
        by [negation]
    QED.
```

will only be added to the hypothesis context if the `Statement`, `(P and (P => Falsity)) => Falsity` is valid and the `Proof` is indeed a proof for `(P and (P => Falsity)) => Falsity`.

- **Comments**: comments are in `ML` style. Everything between `(*` and `*)` is a comment.

Examples

These are shortened proofs. For complete proofs and more sample proofs, extra/sample_proofs.

Law of Excluded Middle

```
Signatures:
    P : prop ;

Definitions:
    [DNE] : ((P => Falsity) => Falsity) => P ;
   [not P] : ((P or (P => Falsity)) => Falsity) => (P => Falsity) ;
    [not not P] : ((P or (P => Falsity)) => Falsity)
                   => ((P => Falsity) => Falsity) ;

Theorem [not not (P or not P)]:
    Statement: ((P or (P => Falsity)) => Falsity ) => Falsity
    Proof:
      assume [not (P or not P)] : (P or (P => Falsity)) => Falsity .
        we know [not P] : P => Falsity
            because [not P] with ([not (P or not P)]) .
        we know [not not P] : (P => Falsity) => Falsity
```

```
            because [not not P] with ([not (P or not P)]) .
        we know [P] : P because [DNE] with ([not not P]) .
        we know [negation] : Falsity because [not P] with ([P]) .
        by [negation]
    QED.

Theorem [excluded middle]:
    Statement: P or (P => Falsity)
    Proof:
        we get [New DNE] :
            (((P or (P => Falsity)) => Falsity) => Falsity)
                => (P or (P => Falsity))
        instantiating [DNE] with (P is (P or (P => Falsity))) .
        [New DNE] with ([not not (P or not P)])
    QED.
```

Note that many aspects of the proof are stylistic. For instance, to make it clearer, the proof for `[not not (P or not P)]` ends with `by [negation]`. This could be compacted since the last step is redundant. e.g.

```
    we know [negation] : Falsity because [not P] with ([P]) .
    by [negation]
```

becomes

```
    [not P] with ([P])
```

However, this is not as clear as stating `by [negation]`. Another example of this can be seen with `[New DNE] with ([not not (P or not P)])`. Here, no final `by` step is used. If we wanted to make it clearer, we could annotate it with a `therefore` clause.

```
    [New DNE] with ([not not (P or not P)])
```

becomes

```
   [New DNE] with ([not not (P or not P)]) therefore P or (P => Falsity)
```

Additionally, `hypotheses` generally have an optional annotation. In `we know` and `we get` clauses, this annotation is compulsory. In others, it might not be. For instance:

```
    by [negation]
```

becomes

```
    by [negation] : Falsity
```

with the optional annotation.

Involution of Reversing a List

```
Signatures:
    append : nat list -> nat list -> nat list ;
    rev    : nat list -> nat list ;

Definitions:
   [append nil] : forall xs : nat list . append [] xs = xs : nat list ;
    [append xs]  : forall xs : nat list .
                    forall x : nat .
                        forall ys : nat list .
                   append (x::xs) ys = x :: append xs ys : nat list ;
    [rev nil]    : rev [] = [] : nat list ;
    [rev xs]     : forall xs : nat list .
                    forall x : nat .
                rev (x :: xs) = append (rev xs) (x :: []) : nat list ;
    [rev lemma]  : forall xs : nat list .
                    forall x : nat .
                 rev (append xs (x::[])) = x :: (rev xs) : nat list ;

Theorem [involution of rev] :
    Statement: forall xs : nat list . rev (rev xs) = xs : nat list
    Proof:
        by induction on list :

        case [] :
            equality on ([rev nil])

        case (hd :: tl) : [inductive hypothesis] .
            we know [step 1] : rev (hd :: tl) =
                               append (rev tl) (hd::[]) : nat list
            because [rev xs] with (tl,hd).

            we know [step 2] : rev (append (rev tl) (hd::[])) =
                                hd :: (rev (rev tl)) : nat list
            because [rev lemma] with (rev tl,hd) .

          equality on ([step 1], [step 2], [inductive hypothesis])
    QED.
```

Like the previous example, here we too have optional annotations; the inductive hypothesis `[inductive hypothesis]` doesn't need a proposition annotation. It is just there to make things clearer.

Note that even though it's redundant, if you do decide to give it an annotation, it must match against the expected proposition. If it doesn't, then the proof will fail on the incorrect annotation and give you the corresponding error message.