reasonEq: Tutorial
(v0.8.0.0)

Andrew Butterfield

March 7, 2024

Contents

1 Prerequisites

reasonEq has been installed and started for at least the first time according to instructions in the top-level README file.

You should have seen a transcript similar to this (precise details depend on your OS - that below was on macOS):

```
:- req
starting REPL...
Running user mode, default initial state.
Creating app. dir.: /Users/yourusername/.reasonEq
Creating workspace : /Users/yourusername/.../MyReasonEq
appFP = /Users/yourusername/.reasonEq
projects:
*MyReasonEq|/Users/yourusername/.../MyReasonEq
```

```
Creating /Users/yourusername/.../MyReasonEq
Creating /Users/yourusername/../MyReasonEq/project.req
Project Name: MyReasonEq
Project Path: /Users/yourusername/TEST/MyReasonEq
Loading...
Welcome to the reasonEq 0.6.9.0 REPL
Type '?' for help.

MyReasonEq.>
```

You are now using the "Top-Level" command line interface.

2 Getting Help

Requesting help by typing `help` or `?` results in:

```
quit -- exit
?,help -- this help text
?,help <cmd> -- help for <cmd>
sh -- show parts of the prover state
set -- set parts of the prover state
new -- generate new theory items
N -- new proof
r -- return to live proof
save -- save prover state to file
load -- load prover state from file
svc -- save conjectures
ldc -- load conjectures
Assume -- assume conjecture is law
Demote -- demote law to conjectures
b -- builtin theory handling
classify -- activate classifier
```

More help on a specific command is given by supplying it to help, so, for example, typing `? sh` results in:

```
sh w -- show workspace info
sh X -- show settings
sh s -- show logic signature
sh t -- show theories
sh L -- show laws
sh L -u -- show variable uniqueness
sh k -- show known names
sh T -- show 'current' theory
sh c -- show current conjectures
sh c -u -- show variable uniqueness
sh p -- show current (live) proof
sh P -- show completed theory proofs
sh P * -- show all completed proofs
sh P <nm> -- show proof transcript for <nm>
```

‘

3 Builtin Theories

Currently it is not possible for the user to create new theories, or add new axioms to existing theories. Instead, some builtin theories have been defined, but they are not "installed" by default.

The `b` command allows the installation and checking of builtin theories. Entering `? b` results in the following:

```
b e -- list all existing builtin theories
b i -- list all installed theories
b I <name> -- install builtin theory <name>
           -- fails if theory already installed
b R <name> -- reset builtin theory <name>
           -- replaces already installed theory by builtin version
                                      (a.k.a. 'factory setting')
b U <name> -- update builtin theory <name>
           -- adds in new material from builtin version
           -- asks user regarding revisions to existing material
b F <name> -- force-update builtin theory <name>
           -- adds in new and revised material from builtin version
           -- does not ask user to confirm revisions
```

Issuing the command `b e` should result in something like:

```
Equiv ; Not ; Or ; And ; AndOrInvert ; Implies ; Equality ;
ForAll ; Exists ; UClose ; UTPBase
Remember to update Dev.devKnownBuiltins with new builtins.
```

For this tutorial we need theory "Equiv" to be installed using `b I`, and checked using `b i`:

```
MyReasonEq.> b I Equiv
MyReasonEq.Equiv*> b i
Equiv
MyReasonEq.Equiv*>
```

The "Equiv" theory is installed and is now the current working theory. The prompt now includes the current theory name. The asterisk on the prompt indicates that the prover state has been modified, but not yet saved. Save it, just to be safe:

```
MyReasonEq.Equiv*> save
REQ-STATE written to '/Users/yourusername/.../MyReasonEq'.
MyReasonEq.Equiv>
```

Now, ask to see all the known laws, using `sh L` :

```
Theory 'Equiv'
Knowns:
≡ : (𝔹 ⟶ (𝔹 ⟶ 𝔹))
Laws:
   1. ⊤  "true"       true  ⊤
   2. ⊤  "≡_refl"     P  ≡  P  ⊤
   3. ⊤  "≡_assoc"    ((P≡Q)≡R)  ≡  (P≡(Q≡R))  ⊤
   4. ⊤  "≡_symm"     P≡Q≡Q≡P  ⊤
   5. ⊤  "id_subst"   P[x$/x$]  ≡  P  ⊤
Conjectures:
   1. ?  "≡_id"          (true≡Q)  ≡  Q  ⊤
   2. ?  "true_subst"    true[e$/x$]  ≡  true  ⊤
   3. ?  "≡_subst"       (P≡Q)[e$/x$]  ≡  (P[e$/x$]≡Q[e$/x$])  ⊤
AutoLaws:
   i. simps:

  ii. folds:

 iii. unfolds:
```

There are four sections:

Knowns : Identifiers that denote themselves only.

Laws : All available laws. Axioms are marked on the left with ⊤.

Conjectures : All available conjectures, marked on the left with '?'. These need proofs to become theorems.

AutoLaws : Lists of laws that can play specific roles in proof automation.

The use of ⊤ on the right indicates a trivial (true) side-condition.

4 Finding Conjectures

We can concentrate on conjectures using `sh c` :

```
1. "≡_id"          (true≡Q)   ≡  Q   ⊤
2. "true_subst"    true[e$/x$]  ≡  true   ⊤
3. "≡_subst"       (P≡Q)[e$/x$]   ≡   (P[e$/x$]≡Q[e$/x$])   ⊤
```

5 Proving ≡_id

Start a new proof by entering `N 1` (new proof for conjecture 1). You are shown a list of four options:

```
Select proof strategy:
   1. 'red-All':  (true≡Q)  ≡  Q     -->     true
   2. 'red-L2R':  true  ≡  Q     -->     Q
   3. 'red-R2L':  Q     -->     true  ≡  Q
   4. 'red-bth':  true  ≡  Q  --> ? <--  Q
Select sequent by number:
```

Each option corresponds to a different strategy that is applicable to the conjecture. Proof strategies will be described in detail elsewhere, and for now we will simply select the "red-All" strategy by typing '1'. This strategy is simply to try to reduce the whole conjecture down to 'true'. In effect we are going to prove this by reducing the lefthand-side $(true \equiv Q) \equiv Q$ to the righthand-side *true*.

Entering '1' causes the screen to clear (on OS X/Unix at least) and displays something like this:

```
Prover starting...
Max. Match Display = 20
Hide Trivial Matches = False
Hide Trivial Quantifiers = True
Hide Floating Variables = False

Proof for equiv_id
        (true≡Q)  ≡  Q  ⊤
by red-All
 ...


————————————

⊢
(true≡Q)≡Q    ⊤

Focus = []  Target (RHS): true


proof:
```

We see that we are proving conjecture '≡_id' using the 'red-All' strategy. We are told that our target (righthand-side) is 'true'. We see our lefthand-side displayed, in color, and finally we see that we have a new command line prompt `proof:`.

We are now in the Prover command line interface. This has a different set of commands to the top-level one, but still has the same help mechanism:

```
proof: ?

q -- exit
? -- this help text
? <cmd> -- help for <cmd>
ll -- list laws
d -- down
u -- up
m -- match laws
a -- apply match
fe -- flatten equivalences
ge -- group equivalences
b -- go back (undo)
i -- instantiate
s -- switch
h -- to hypothesis
l -- leave hypothesis
c -- clone hyp
```

5.1 Proof Step 1

We start by invoking the 'Matcher' that tries to match the current goal, here $(true = Q) \equiv Q$, against known laws. This is done by typing `m` to produce:

```
Proof for equiv_id
        (true≡Q)  ≡  Q  ⊤
by red-All
 ...
Matches:
7 : "id_subst" ((true≡Q)≡Q)[?x$/?x$]  ⊤ ⟹ ⊤ trivial!2
6 : "≡_symm" ?Q≡?Q≡((true≡Q)≡Q)  ⊤ ⟹ ⊤ trivial!1
5 : "≡_symm" ?P≡((true≡Q)≡Q)≡?P  ⊤ ⟹ ⊤ trivial!2
4 : "≡_symm" ((true≡Q)≡Q)≡?Q≡?Q  ⊤ ⟹ ⊤ trivial!4
3 : "≡_symm" (true≡Q)  ≡  Q  ⊤ ⟹ ⊤ ≡[3,4]
2 : "≡_symm" Q  ≡  (true≡Q)  ⊤ ⟹ ⊤ ≡[1,2]
1 : "≡_assoc" true  ≡  (Q≡Q)  ⊤ ⟹ ⊤ ≡lhs
————————————

⊢
(true≡Q)≡Q    ⊤

Focus = []  Target (RHS): true


proof:
```

The matches are ranked and displayed with the highest rank at the bottom (closest to where the action is). The matched law-name is shown, followed by what would result if that law was applied. The notation $\top \implies \top$ is relating side-conditions from the conjecture to those associated with the laws. Currently all side-conditions are true.

The law "≡_symm" is matched in 5 different ways. There are strange comments at the end (ie., `trivial!2`, `≡[3,4]`). We ignore these for now.

In our case, we see that it matched, amongst other things, against the lefthand-side of the '≡_assoc' law and is giving back the righthand-side. This is what we want so we request that match 1 be applied, using command `a 1` (or `a1`).

This results in:

```
Proof for equiv_id
        (true≡Q)  ≡  Q  ⊤
by red-All
(true≡Q)≡Q
   = 'match-lhs ≡_assoc@[]'
 ...

____________

⊢
true≡(Q≡Q)     ⊤

Focus = []  Target (RHS): true


proof:
```

We see that the goal has changed, and also that we have the start of a proof transcript showing the the original goal was transformed by a match with the *lefthand side* of '≡_assoc' law at the top-level (`@[]`).

5.2 Proof Step 2

We now want to focus attention on the $Q \equiv Q$ sub-part of the goal. It is the second argument to the top-level $\equiv$ operator, so we want to move the focus down to that 2nd argument. We do this using the command `d2` (down 2):

```
Proof for equiv_id
        (true≡Q)  ≡  Q  ⊤
by red-All
(true≡Q)≡Q
   = 'match-lhs ≡_assoc@[]'
 ...


————————————

⊢
true≡(Q≡Q)    ⊤

Focus = [2]  Target (RHS): true


proof:
```

Now we see the significance of the purple colour — it signifies that we are focussed in on a part of the overall goal. Note also that the value of `Focus` has changed from `[]` to `[2]`. This is a list that describes how to get from the top-level of the goal down to the sub-term that is the current focus. We now want to see what this matches against, so we issue the match command `m` once more:

```
Proof for equiv_id
        (true≡Q)  ≡  Q  ⊤
by red-All
(true≡Q)≡Q
   = 'match-lhs ≡_assoc@[]'
 ...
Matches:
6 : "id_subst" (Q≡Q)[?x$/?x$]  ⊤ ⟹ ⊤ trivial!2
5 : "≡_symm" ?Q≡?Q≡(Q≡Q)  ⊤ ⟹ ⊤ trivial!1
4 : "≡_symm" ?P≡(Q≡Q)≡?P  ⊤ ⟹ ⊤ trivial!2
3 : "≡_symm" (Q≡Q)≡?Q≡?Q  ⊤ ⟹ ⊤ trivial!4
2 : "≡_symm" Q  ≡  Q  ⊤ ⟹ ⊤ ≡[3,4]
1 : "≡_refl" true  ⊤ ⟹ ⊤ *

————————————

⊢
true≡(Q≡Q)    ⊤

Focus = [2]  Target (RHS): true


proof:
```

Here the first match is against all of the law ‘≡_refl‘, so we use `a 1` to apply it:

```
Proof for equiv_id
        (true≡Q)  ≡  Q  ⊤
by red-All
(true≡Q)≡Q
   = 'match-lhs ≡_assoc@[]'
true≡(Q≡Q)
   = 'match-all ≡_refl@[2]'
 ...


————————————

⊢
true≡true    ⊤

Focus = [2]  Target (RHS): true
```

Note that we are still focussed at the same place. Here we matched *all* the law ‘≡_refl‘ at the second component of the top-level goal (`@[2]`).

5.3 Proof Step 3

To complete, we want to return to the top-level, so we issue the up-command `u` :

```
Proof for equiv_id
        (true≡Q)  ≡  Q  ⊤
by red-All
(true≡Q)≡Q
   = 'match-lhs ≡_assoc@[]'
true≡(Q≡Q)
   = 'match-all ≡_refl@[2]'
 ...


————————————

⊢
true≡true    ⊤

Focus = []  Target (RHS): true
```

Now we match and see the following matches:

```
Proof for equiv_id
        (true≡Q)  ≡  Q  ⊤
by red-All
(true≡Q)≡Q
   = 'match-lhs ≡_assoc@[]'
true≡(Q≡Q)
   = 'match-all ≡_refl@[2]'
 ...
Matches:
6 : "id_subst" (true≡true)[?x$/?x$]  ⊤ ⟹ ⊤ trivial!2
5 : "≡_symm" ?Q≡?Q≡(true≡true)  ⊤ ⟹ ⊤ trivial!1
4 : "≡_symm" ?P≡(true≡true)≡?P  ⊤ ⟹ ⊤ trivial!2
3 : "≡_symm" (true≡true)≡?Q≡?Q  ⊤ ⟹ ⊤ trivial!4
2 : "≡_symm" true  ≡  true  ⊤ ⟹ ⊤ ≡[3,4]
1 : "≡_refl" true  ⊤ ⟹ ⊤ *
————————————

⊢
true≡true    ⊤

Focus = []  Target (RHS): true
```

We want the same law as the previous step, so we apply the first (`a1`).

```
[proof: a1

Proof for equiv_id
        (true≡Q)  ≡  Q  ⊤
by red-All
(true≡Q)≡Q
   = 'match-lhs ≡_assoc@[]'
true≡(Q≡Q)
   = 'match-all ≡_refl@[2]'
true≡true
   = 'match-all ≡_refl@[]'

Proof Complete
MyReasonEq.Equiv*>
```

The proof is complete, so this reported, including a proof transcript, and we exit the proof command-line and return to the top-level command line.

5.4 The aftermath

Now take a look at the theory using `sh T`:

```
Theory 'Equiv'
Knowns:
≡ : (𝔹 ⟶ (𝔹 ⟶ 𝔹))
Laws:
   1. ⊤  "true"       true   ⊤
   2. ⊤  "≡_refl"     P  ≡  P   ⊤
   3. ⊤  "≡_assoc"    ((P≡Q)≡R)  ≡  (P≡(Q≡R))   ⊤
   4. ⊤  "≡_symm"     P≡Q≡Q≡P   ⊤
   5. ⊤  "id_subst"   P[x$/x$]  ≡  P   ⊤
   6. ■  "≡_id"       (true≡Q)  ≡  Q   ⊤
Conjectures:
   1. ❓  "true_subst"  true[e$/x$]  ≡  true   ⊤
   2. ❓  "≡_subst"     (P≡Q)[e$/x$]  ≡  (P[e$/x$]≡Q[e$/x$])   ⊤
AutoLaws:
   i. simps:

  ii. folds:

 iii. unfolds:
```

We see the proven conjecture (≡_id) has become a law. This law differs from the axioms in that it is marked on the left with ■ instead of '⊤' to show that it is a theorem with a proof, rather than an axiom.

6 Next Steps.

Try installing theory 'Not' and proving its conjectures. Then 'Or', 'And', 'AndOrInvert', and 'Implies'.