The assertion client manual

quantifier.cloud

Contents

1 Introduction

This document is a client manual for assertion™: a resolution based, automated first-order predicate logic prover featuring the following three key properties:

(i) it is *sound*,

(ii) it is *refutationally complete*,

(iii) the properties (i) and (ii) have been *formally verified* in Isabelle/HOL.

The practical relevance of soundness is immediate: the prover does not confirm any assertion that is not generally valid, *i.e.* not a tautology. On the other hand, practical relevance of refutational completeness is more subtle: if a formula is a tautology then the prover shall eventually confirm that (*i.e.* refute satisfiability of its negation), yet for a vast amount of tautologies such *direct* refutations tend to demand resources which are a good deal beyond of what is attainable at present. More precisely, due to various limitations inherent to physical hardware (such as *RAM* shortage), also for refutationally complete provers there **cannot be a warranty** that a given tautology gets eventually confirmed on **any** physical system. Nonetheless, due to refutational completeness the more effective the underlying physical system the more likely the prover can find a proof for a tautology.

Another notable feature of assertion is that it takes over the adjustment of 'technical' parameters steering internal reasoning processes. The more important may become 'logical adjustments' of an assertion such as providing a hint to the prover by means of an additional assumption which is actually a consequence of other assumptions, or specialising a universally quantified premise in accord to a particular conclusion, or removing a premise from an implication if this premise not only turns out to be unnecessary but also to considerably slow down the reasoning process aiming to reach some particular conclusion. In other words, based on the key properties (i)–(iii) above, the approach to confirming a tautology, whose direct confirmation would demand unattainable resources, is to obtain an indirect confirmation as a consequence of validity of adjusted formulas which in turn can be confirmed by the prover using computational resources at hand.

This manual is structured as follows: next short section explains how the assertion-client can be deployed for processing a file by the assertion-server. Section 3 in detail describes the syntax rules for writing an assertion-file. Section 4 first gives some introductory examples of how the assertion-prover can effectively be applied using a client. Then Section 4.1 describes the structure of proof files that can be generated for confirmed assertions and also outlines how generated proofs can in turn be replayed by interactive provers supporting higher-order logics. Further, Section 4.2 gives an example of how exponential blow-ups can be mitigated. Lastly, Section 4.3 contains a few concluding remarks.

2 Deploying the assertion-client

The prover is running as a cloud service that processes assertion-files sent using the client `assertion/Client.class` which comprises a `java`-package.

The client gets easily deployed on any system where a stable internet connection, a *J*ava *R*untime *E*nvironment (JRE) as well as a terminal console (a command prompt) are accessible:

– place the folder `assertion` containing `Client.class` in a folder `<dir>` on your system, typing subsequently in a terminal console

```
cd <dir>
java assertion.Client <path to an assertion-prover file>
```

to request the server to process `<an assertion-prover file>`.

For instance, if the file `misc.ap` resides in the folder `<dir>/examples` then

```
cd <dir>
java assertion.Client examples/misc.ap
```

(on some systems one has to use backslashes, *i.e.* `examples\misc.ap`) requests the server to process the content of `misc.ap` which results in a response sent back to the client. When the server responds to a syntactically correct assertion contained in `<an assertion-prover file>` (*cf.* Section 3 below), the answer can be either

`confirmed` – *i.e.* the respective formula is an (equational) tautology,
or
`rejected` – *i.e.* an (equational) model for the negated formula exists.

Without a subscription[*], the current limit of about 30 seconds for processing one assertion-file applies, *i.e.* the connection to the server gets closed straight when the limit is exceeded. Note that **no data** is collected by the server except basic connection related statistics.

3 The syntax

3.1 The basic structure of an assertion-file

A file may contain a series of assertions separated by semicolons as follows

```
 assertion "<assert-id>" ["<comments>"] : <formula>;
 assertion "<assert-id>" ["<comments>"] : <formula>;
...
 assertion "<assert-id>" ["<comments>"] : <formula>
```

where `<assert-id>` and `<comments>` are placeholders for random strings without line breaks. It is further advisable to avoid special characters in `<assert-id>` and make sure that no two assertions get the same name attached. These recommendations become actually more valuable when proof files shall be generated (*cf.* Section 4.1).

As there is nothing to separate when a file contains only one assertion, no semicolons shall accordingly be used in such cases.

Noting moreover that comments sections are optional, the following line:

```
 assertion "<assert-id>" : <formula>
```

is a template for an eligible file comprising one assertion without comments.

3.2 Formula syntax

The syntax of formulas resembles Isabelle/HOL object-logical syntax: the negation operator is denoted by `~`, the existential quantifier – by `EX <vars>. ...`, the universal – by `ALL <vars>. ...` and so on, where `<vars>` stands for a non-empty list of variable identifiers `<var-id>`.

More precisely, syntactically correct formulas are strings derivable from the non-terminal symbol `<formula>` defined by the below grammar that implements the usual precedence as well as the left/right-associativity conventions (where ϵ denotes the empty string):

[*]see client/README for details.

```
<formula>  ::=  ALL <vars>. <formula>
            |   EX <vars>. <formula>
            |   <$F_1$>

    <$F_1$>  ::=  <$F_2$>  -->  <$F_1$>
            |   <$F_2$>  <->  <$F_1$>
            |   <$F_2$>

    <$F_2$>  ::=  <$F_3$><$F'_2$>

    <$F'_2$>  ::=   &  <$F_3$><$F'_2$>
            |    |  <$F_3$><$F'_2$>
            |       $\epsilon$

    <$F_3$>  ::=   ~  <$F_4$>
            |   <$F_4$>

    <$F_4$>  ::=   True
            |    False
            |    <pred-id> ( <terms1> )
            |    <pred-id>
            |    <term> <pred-infix-id> <term>
            |    ( <formula> )

 <terms1>  ::=   <term> , <terms1>
            |    <term>

  <terms>  ::=   <terms1>
            |     $\epsilon$

   <term>  ::=   <funct-id> ( <terms> )
            |    <var-id>
            |    <term'> <funct-infix-id> <term'>

  <term'>  ::=   <funct-id> ( <terms> )
            |    <var-id>
            |    ( <term'> <funct-infix-id> <term'> )
```

Furthermore, there are the following rules concerning predicate, function and variable identifiers, *i.e.* `<pred-id>` , `<pred-infix-id>` , `<funct-id>` , `<funct-infix-id>` and `<var-id>` :

– `<funct-id>` and `<var-id>` have to consist of **one lower case** alphanumerical symbol, optionally followed by **another lower case** alphanumerical symbol:

```
a, b, c, f, g1, xa, ...
```

so that `g(x)` and `fa(h1(x1), x2)` and `f(5())` are examples of syntactically correct terms, whereas *e.g.* `fgh(x)` has a function identifier with more than two symbols and hence syntactically incorrect

– `<pred-id>` has to consist of **one upper case letter**, optionally followed by **another alpha-numerical symbol**: `A, B, C, DX, Gx, B1, M2 ...` so that `P(g(x))` and `A1(y)` and `Qa(xb(), h(x2))` are examples of syntactically correct formulas (note moreover that parentheses have to be omitted for propositional (rank 0) predicates, *i.e.* `P & Q` is a syntactically correct propositional conjunction as opposed to `P() & Q()`)

– the **unambiguous rank** restriction applies generally to function as well as to predicate symbols, *i.e.* `P(x) | P(x, x)` and `P(f(y)) --> Q(f(y, y))` are examples of **incorrect** formulas since the predicate `P` and the function `f` occur there with two ranks: 1 and 2

– the admissible infix predicate symbols `<pred-infix-id>` are:

```
=  <  >  <=  =>  >=  <<  >>  ==  <>
```

(note that `=` is interpreted as the equality on domain values, whereas all other symbols are *per se* uninterpreted, *i.e.* `ALL a. EX b. a = b` will for example be confirmed by the prover as it is a basic *equational* tautology signifying that for any value in the evaluation domain there is an equal value whereas *e.g.* `ALL a. EX b. a == b` will be rejected because there are evidently models for `EX a. ALL b. ~ a == b`

– the admissible infix function symbols `<funct-infix-id>` are:

```
/  %  @  +  *  -  !  ++  --  **  ->  <-
```

and none of these is *per se* interpreted, *i.e.* `ALL a b. a + b = b + a` is for instance *not* a tautology

– note that **none** of the above infix symbols is a lexical separator such that *e.g.* `a+b = a + b` is *syntactically incorrect* because `a+b` gets read as one lexical token, whereas *e.g.* `a *(b + c)= a * (b + c)` is possible because parentheses do separate lexical tokens alike the whitespace character.

4 assertion-server at work

There are usually various logical paths leading to a confirmation that a formula is indeed a tautology. However, the amount of required resources may greatly vary in accord to the taken approach. An example is the associativity property of the join-operator on partial orders:

```
assertion "join-assoc" :
(ALL a. a <= a) &
(ALL a b c. a <= b & b <= c --> a <= c) &
(ALL a b. a <= b --> b <= a --> a = b) &
(ALL a b. a <= a + b) &
(ALL a b. b <= a + b) &
(ALL a b x. a <= x --> b <= x --> a + b <= x) -->
(ALL a b c. (a + b) + c = a + (b + c))
```

In short: if `<=` is a partial order and `+` assigns to any two elements their least upper bound then `+` must be associative. Although with a powerful CPU and lots of patience a confirmation for this tautology shall eventually be reached, one can alternatively utilise that with `join-assoc`, the antisymmetry assumption

```
ALL a b. a <= b --> b <= a --> a = b
```

has no other purpose than to infer `ALL a b c. (a + b) + c = a + (b + c)` from the 'left/right associativity':

```
ALL a b c. (a + b) + c <= a + (b + c) & a + (b + c) <= (a + b) + c
```

Hence, there is actually no need to involve specifically the equality here, *i.e.* one can replace the two occurrences of `=` in `join-assoc` by *e.g.* `==` which is an infix notation for a *per se* uninterpreted relation as opposed to `=` denoting the equality as pointed out in the preceding section. This generalisation greatly simplifies the prover's task. It is however even more productive to put back the antisymmetry assumption and consider for the beginning

```
assertion "join-assoc-lr" :
(ALL a. a <= a) &
(ALL a b c. a <= b & b <= c --> a <= c) &
(ALL a b. a <= a + b) &
(ALL a b. b <= a + b) &
(ALL a b x. a <= x --> b <= x --> a + b <= x) -->
(ALL a b c. (a + b) + c <= a + (b + c) &
            a + (b + c) <= (a + b) + c)
```

which gets confirmed by the prover fairly fast alike the following, thus only formally weaker version of `join-assoc`

```
assertion "join-assoc-weak" :
(ALL a. a <= a) &
(ALL a b c. a <= b & b <= c --> a <= c) &
(ALL a b. a <= b --> b <= a --> a = b) &
(ALL a b. a <= a + b) &
(ALL a b. b <= a + b) &
(ALL a b x. a <= x --> b <= x --> a + b <= x) &
(ALL a b c. a + (b + c) <= (a + b) + c &
            (a + b) + c <= a + (b + c)) -->
(ALL a b c. (a + b) + c = a + (b + c))
```

which resorts to the left/right associativity as an interpolating condition. Altogether, `join-assoc-lr` and `join-assoc-weak` entail `join-assoc`.

As a particular consequence, one may safely add the join-associativity to the premises for reasoning upon further properties of partial orders with a join-operator whenever it is expedient:

```
assertion "..." :
(ALL a. a <= a) &
(ALL a b c. a <= b & b <= c --> a <= c) &
(ALL a b. a <= b --> b <= a --> a = b) &
(ALL a b. a <= a + b) &
(ALL a b. b <= a + b) &
(ALL a b x. a <= x --> b <= x --> a + b <= x) &
(ALL a b c. (a + b) + c = a + (b + c)) -->
 ...
```

The principle gets frequently applied *e.g.* in the algebraic group setting. It is particularly well-known that the associativity, the 'left unit' and the 'left inverse' assumptions entail the 'right unit' and the 'right inverse' properties:

```
assertion "right unit" :
(ALL a b c. (a * b) * c = a * (b * c)) &
(ALL a. 1() * a = a) &
(ALL a. i(a) * a = 1()) -->
(ALL a. a * 1() = a);

assertion "right inverse" :
(ALL a b c. (a * b) * c = a * (b * c)) &
(ALL a. 1() * a = a) &
(ALL a. i(a) * a = 1()) -->
(ALL a. a * i(a) = 1())
```

From the automated reasoning perspective it is nonetheless often beneficial to add these conditions to the assumptions:

```
assertion "..." :
(ALL a b c. (a * b) * c = a * (b * c)) &
(ALL a. 1() * a = a) &
(ALL a. i(a) * a = 1()) &
(ALL a. a * 1() = a) &
(ALL a. a * i(a) = 1()) -->
 ...
```

because various instances of `ALL a. a * 1() = a` and `ALL a. a * i(a) = 1()` tend to arise as subgoals in the course of deriving more advanced group properties. Hence keeping these assumptions might save a lot of additional and redundant work for any automated prover.

4.1 Generating proofs[*]

It is often important to have an explicit justification *why* an assertion has been confirmed, *i.e.* a detailed proof. To achieve that, the prover has to keep track of the performed inference steps and, as a side effect, to consume slightly more resources.

More precisely, a request using the keyword `prove` such as

```
prove assertion "disj_elim" :
(A | B) & (A --> C) & (B --> C) --> C
```

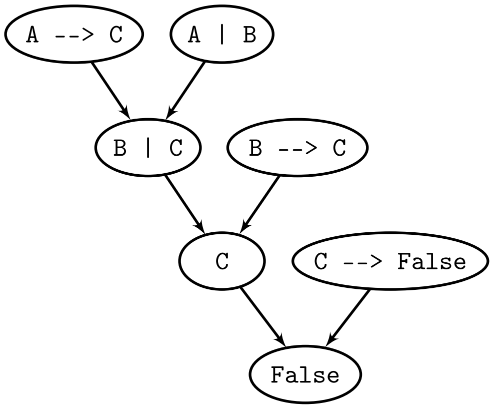
not only brings the server about to return a confirmation for `disj_elim` but also to transfer a proof file (named `disj_elim.dot` in this case) or proof files `<assert-id>_1.dot`, `<assert-id>_2.dot`, ... (this will be explained later on in detail) back to the client. The suffix `.dot` emphasises that proof files support the `dotty` graph visualisation format (*cf.* graphviz.org) and the basic structure of all proof files is the same as explained using `disj_elim.dot` next.

4.1.1 The file `disj_elim.dot`

Any proof file comprises two parts: a `dotty`-format description followed by a textual description. This especially means that typing

```
   dotty disj_elim.dot
```

makes the `dotty` tool display the tree structure



but in general this output will be a directed acyclic graph. The second part of the `disj_elim.dot` file contains the textual, assertion-prover specific description of the above tree structure:

```
>>> assume
   #1: C --> False
   #2: A --> C
```

[*]The functionality is subject to subscription, see client/README for details.

```
   #3: A | B
   #4: B --> C
>>> show: False
>>> proof:
   from #3 and #2 have #5:  B | C
   from #5 and #4 have #6:  C
   from #6 and #1 have False
>>> qed
```

where the basic proof inference steps of the form

> `...`
>
> `from` $\#L_1$ `and` $\#L_2$ `have` $\#L_3$`: <formula>`
>
> `...`

hold the following precise meaning:

> *if there is a model for the two formulas referred to by the labels L_1 and L_2 then it is also a model for* `<formula>` *to which the label L_3 shall refer.*

Note that L_3 gets always omitted in the last inference of `False` as there is no need to retrieve this formula: a model for `False` does not exist, hence there cannot be a model that satisfies all of the formulas in the `assume` section, *i.e.* the conjunction of the listed assumptions can be regarded as refuted. Since to any (equational) model for the negation of the original assertion there exists a model that satisfies all of the listed assumptions, the assertion must be a first-order (equational) tautology.

In this particular example, from the unsatisfiability of the assumptions

```
   #1: C --> False
   #2: A --> C
   #3: A | B
   #4: B --> C
```

follows that the assertion `disj_elim` is a (in this case propositional) tautology.

Furthermore, it should not pose any notable challenge to carry the above proof over to a proof assistant providing a few automated tactics such as Lean:

```
theorem disj_elim_neg (A B C : Prop)
(a1: C -> False)
(a2: A -> C)
(a3: Or A B)
(a4: B -> C)
 : False := by
 have a5: Or B C := by grind
 have a6: C := by grind
 exact a1 a6
```

Note that one might try to interactively replace `grind` by `apply?` for obtaining a detailed chain of Lean inference rules that can be used to resolve the respective subgoal without automation.

4.1.2 A more advanced example

The following query to the server

```
prove assertion "eq_example1" :
(ALL a. R(a, f(a))) & (ALL b. EX a. f(a) = b) -->
(ALL b. EX a. R(a, b))
```

returns the proof file `eq_example1.dot` containing the description:

```
>>> assume there exist sk_103 and sk_104 such that
   #1: ALL a b c. a = b & c = b --> a = c
   #2: ALL a. f(sk_103(a)) = a
   #3: ALL a. a = a
   #4: ALL a. R(a, f(a))
   #5: ALL a. R(a, sk_104) --> False
   #6: ALL a b c d. R(a, b) & a = c & b = d --> R(c, d)
>>> show: False
>>> proof:
   from #5 and #6 have #7:  ALL a b c. R(a, b) & a = c & b = sk_104
                              --> False
   from #4 and #7 have #8:  ALL a b. a = b & f(a) = sk_104 --> False
   from #2 and #8 have #9:  ALL a. sk_103(sk_104) = a --> False
   from #2 and #1 have #10:  ALL a b. a = b --> a = f(sk_103(b))
   from #3 and #10 have #11:  ALL a. a = f(sk_103(a))
   from #9 and #11 have False
>>> qed
```

First of all note that the proof summarises a *mechanised* reasoning process and in particular does not necessarily reflect the order of individual inferences: `#11` for instance could have been inferred prior to `#9`. One may also notice that the generated proof is not the shortest: the last three inference steps could for example be replaced by a single step.

Second, the proof makes use of two so-called *Skolem constants*, namely `sk_103` (rank 1) and `sk_104` (rank 0). Such extra symbols basically emerge out of free variables and existential quantifications occurring in a formula that is in the negation normal form.

The naming of a Skolem constant has no extra purpose than to keep it apart from all other function symbols, *i.e.* one clearly would not change anything significant by writing *e.g.*

```
>>> assume there exist sk_1 and sk_2 such that ...
```

subsequently renaming `sk_103` to `sk_1` and `sk_104` to `sk_2` across both: the assumptions and the proof. However, replacing `sk_103` by *e.g.* `f` or `sk_104` would break the proof.

It has moreover to be pointed out that a free variable in a formula, say `x`, would appear in a generated proof as a Skolem constant `sk_<some index>`

whereas a constant, *e.g.* `x()`, in place of `x` would by contrast retain its name appearing in a generated proof as `x()`.

Although `eq_example1` comprises a first-order formula, its proof is beyond the first-order logic due to the (meta-)quantified Skolem constant `sk_103` referring to some *function* with the property `ALL a. f(sk_103(a)) = a` (*cf.* the assumption #2).

Taking particular advantage of the *Isar* proof language, the above proof can be quite conveniently carried over to Isabelle/HOL:

```
lemma eq_example1 :
"(ALL a. R(a, f(a))) & (ALL b. EX a. f(a) = b) -->
 (ALL b. EX a. R(a, b))"                         (is "?P")
proof(rule ccontr)
  assume "~ ?P"
  then obtain sk_103 and sk_104 where
   1: "ALL a b c. a = b & c = b --> a = c" and
   2: "ALL a. f(sk_103(a)) = a" and
   3: "ALL a. a = a" and
   4: "ALL a. R(a, f(a))" and
   5: "ALL a. R(a, sk_104) --> False" and
   6: "ALL a b c d. R(a, b) & a = c & b = d --> R(c, d)"
    by metis
  show False
  proof-
   from 5 and 6 have 7:
   "ALL a b c. R(a, b) & a = c & b = sk_104 --> False" by metis

   from 4 and 7 have 8:
   "ALL a b. a = b & f(a) = sk_104 --> False" by metis

   from 2 and 8 have 9:
   "ALL a. sk_103(sk_104) = a --> False" by metis

   from 2 and 1 have 10:
   "ALL a b. a = b --> a = f(sk_103(b))" by metis

   from 3 and 10 have 11:
   "ALL a. a = f(sk_103(a))" by metis

   from 9 and 11 show False by metis
 qed
qed
```

which demands only few explanations.

- The instruction `(is "?P")` deploys a pattern matching mechanism in order to make `?P` a shorthand for the statement of the lemma to refer to it within the following proof.

- The command `proof(rule ccontr)` initiates a proof by contradiction allowing us to assume the negation of the statement, done by `assume "~ ?P"`.

- `then obtain sk_103 and sk_104 where ...` essentially asserts that the existence of a function `sk_103` and a constant `sk_104` satisfying `1 -- 6` follows from `~ ?P`, which is immediately confirmed `by metis` (one of the automated tactics provided by Isabelle/HOL). It is moreover important to note that in cases without any Skolem constants the line
  `then obtain ... and ... and ... where`
  simplifies to
  `then obtain`

- Having the Skolem constants and the assumptions now in place, it in principle remains to replay the inference steps of the generated proof. This process gets initiated by
  `show False`
  `proof-`

- Note that the last inference step `from 9 and 11 show False by metis` deviates from the generated proof by using the keyword `show` instead of `have`. Moreover, note that the first `qed` concludes replaying the generated refutational proof whereas the second `qed` concludes the entire proof of the lemma `eq_example1`.

Summing up, *the generated proof* of the assertion `eq_example1` can be regarded as checked by Isabelle/HOL. Note that `metis` is a particularly powerful automated tactic so that one can actually be far more concise:

```
lemma eq_example1 :
"(ALL a. R(a, f(a))) & (ALL b. EX a. f(a) = b) -->
 (ALL b. EX a. R(a, b))"
by metis
```

thereby concealing however which basic inference steps were taken.

4.1.3 Yet another example

Assertions can internally get split to simplify reasoning. From the proof generation perspective this means that processing an assertion may occasionally result in a series of proof files `<assert-id>_1.dot`, ..., `<assert-id>_N.dot`. For instance, the following query

```
prove assertion "eq_example2" :
(ALL a b. R(a, f(b)) <-> a = b) -->
(ALL a. R(a, f(a))) & (ALL a b. f(a) = f(b) --> a = b)
```

produces two files: `eq_example2_1.dot` and `eq_example2_2.dot` with the former containing the textual description

```
>>> assume there exists sk_103 such that
   #1: ALL a. a = a
   #2: R(sk_103, f(sk_103)) --> False
   #3: ALL a b. a = b --> R(a, f(b))
>>> show: False
>>> proof:
   from #2 and #3 have #4:  sk_103 = sk_103 --> False
   from #4 and #1 have False
>>> qed
```

and the latter containing

```
>>> assume there exist sk_103 and sk_104 such that
   #1: ALL a b c d. R(a, b) & a = c & b = d --> R(c, d)
   #2: f(sk_103) = f(sk_104)
   #3: sk_103 = sk_104 --> False
   #4: ALL a b. R(a, f(b)) --> a = b
   #5: ALL a. a = a
   #6: ALL a b. a = b --> R(a, f(b))
>>> show: False
>>> proof:
   from #5 and #6 have #7:  ALL a. R(a, f(a))
   from #3 and #4 have #8:  R(sk_103, f(sk_104)) --> False
   from #2 and #1 have #9:  ALL a b. R(a, f(sk_103)) & a = b -->
                             R(b, f(sk_104))
   from #8 and #9 have #10:  ALL a. R(a, f(sk_103)) & a = sk_103 -->
                              False
   from #5 and #10 have #11:  R(sk_103, f(sk_103)) --> False
   from #7 and #11 have False
>>> qed
```

such that getting a machine-checked proof is somewhat more elaborate than with a single file like in the preceding example.

Firstly, the proof from `eq_example2_1.dot` shall be captured by means of the following separate lemma in Isabelle/HOL:

```
lemma eq_example2_1 :
  "~((ALL a. a = a) &
     (R(sk_103, f(sk_103)) --> False) &
     (ALL a b. a = b --> R(a, f(b))))"
proof(rule notI, elim conjE)
  assume 1: "ALL a. a = a" and
         2: "R(sk_103, f(sk_103)) --> False" and
         3: "ALL a b. a = b --> R(a, f(b))"
  from 2 and 3 have 4: "sk_103 = sk_103 --> False" by simp
  from 4 and 1 show False by simp
qed
```

That is, the generated refutation of the assumptions `#1, #2, #3` gets actually replayed inside a proof for the *negation* of the *conjunction* of these assumptions. Also note that the basic inference steps are for a change established using only the light weight *Isabelle tactics* `simp` (above) and `fast` (below).

The proof from `eq_example2_2.dot` is treated likewise:

```
lemma eq_example2_2 :
  "~((ALL a b c d. R(a, b) & a = c & b = d --> R(c, d)) &
     (f(sk_103) = f(sk_104)) &
     (sk_103 = sk_104 --> False) &
     (ALL a b. R(a, f(b)) --> a = b) &
     (ALL a. a = a) &
     (ALL a b. a = b --> R(a, f(b))))"
proof(rule notI, elim conjE)
  assume 1: "ALL a b c d. R(a, b) & a = c & b = d --> R(c, d)" and
         2: "f(sk_103) = f(sk_104)" and
         3: "sk_103 = sk_104 --> False" and
         4: "ALL a b. R(a, f(b)) --> a = b" and
         5: "ALL a. a = a" and
         6: "ALL a b. a = b --> R(a, f(b))"
  from 5 and 6 have 7:  "ALL a. R(a, f(a))" by fast
  from 3 and 4 have 8:  "R(sk_103, f(sk_104)) --> False" by fast
  from 2 and 1 have 9:  "ALL a b. R(a, f(sk_103)) & a = b -->
                           R(b, f(sk_104))" by fast
  from 8 and 9 have 10:  "ALL a. R(a, f(sk_103)) & a = sk_103 -->
                             False" by fast
  from 5 and 10 have 11:  "R(sk_103, f(sk_103)) --> False" by fast
  from 7 and 11 show False by fast
qed
```

and the following proof of the original assertion `eq_example2` essentially puts the two preceding lemmas together:

```
lemma eq_example2 :
"(ALL a b. R(a, f(b)) = (a = b)) -->
 (ALL a. R(a, f(a))) & (ALL a b. f(a) = f(b) --> a = b)" (is "?P")
proof(rule ccontr)
  assume "~ ?P"
  then obtain sk_103 and sk_104 where
    "((ALL a. a = a) &
      (R(sk_103, f(sk_103)) --> False) &
      (ALL a b. a = b --> R(a, f(b))))
   |
     ((ALL a b c d. R(a, b) & a = c & b = d --> R(c, d)) &
      f(sk_103) = f(sk_104) &
      (sk_103 = sk_104 --> False) &
      (ALL a b. R(a, f(b)) --> a = b) &
      (ALL a. a = a) &
      (ALL a b. a = b --> R(a, f(b))))" (is "?D1 | ?D2")
    by metis
  thus False
  proof
    assume ?D1
    with eq_example2_1 show False ..
  next
    assume ?D2
    with eq_example2_2 show False ..
  qed
qed
```

(note that the slightly different formulation `R(a, f(b)) = (a = b)` just avoids problems with the symbol `<->` in *Isabelle* and that `..` is also a tactic applied above twice to discard the subgoals).

The main difference to the proof in `lemma eq_example1` is that the `obtain` section above comprises a *disjunction* `|` of *conjunctions* `&` of the assumptions from each of the generated proofs, such that `?D1` and `?D2` consequently refer to the two disjuncts by means of pattern matching.

The theoretical justification for the split into two subgoals `?D1` and `?D2` is in principle based on the previously considered tautology `disj_elim`

```
(A | B) & (A --> C) & (B --> C) --> C
```

having

```
(A | B) & (A --> False) & (B --> False) --> False
```

as a particular consequence. That is, to refute the disjunction `?D1 | ?D2` it is sufficient to refute both, `?D1` and `?D2`, respectively accomplished by the generated proofs in `eq_example2_1.dot` and `eq_example2_2.dot`. More generally, if an assertion gets split into `N` subgoals then accordingly `N` proof files would be generated, together refuting a disjunction `?D1 | ... | ?DN`.

4.2 Mitigating exponential blow-ups

Consider

```
assertion "double negation" :
(ALL x. x = 1() | x = 0()) &
(ALL x. ~ x = f(x)) -->
 f(f(v)) = v
```

which essentially signifies that negating a Boolean value `v` twice results in the same value. Although it is basically clear that the formula is a tautology, from the prover perspective it is not as straight forward because the clause

```
 ALL x. x = 1() | x = 0()
```

yields the relevant instances

```
 v = 1() | v = 0()
 f(v) = 1() | f(v) = 0()
 f(f(v)) = 1() | f(f(v)) = 0()
```

and these tend to contribute very eagerly to the reasoning process which thus has to deal with a large amount of successive clauses that combine all of the above cases. In short, this direct approach creates such a combinatorial overhead that considerable resources become necessary in order to reach a confirmation.

Like in many other situations, generalisation turns out to be very helpful here too. More precisely, nothing essential changes if one does not explicitly prohibit interpretation domains that may contain other values alongside the value(s) of `1()` and `0()`. This can be expressed by means of an extra predicate `B` (for 'Boolean') as follows

```
assertion "double negation generalised" :
(ALL x. B(x) --> B(f(x)) & (x = 1() | x = 0())) &
(ALL x. B(x) --> ~ x = f(x)) -->
(ALL x. B(x) --> f(f(x)) = x)
```

with the effect that it takes only a few seconds to confirm the latter assertion. The reason is that the clause `ALL x. ~ B(x) | x = 1() | x = 0()` (*i.e.* the same as before yet under the premise `B(x)`) by contrast provides for much more controlled derivation of successive clauses.

In line with this should also be mentioned that it is basically worth once more checking if universally quantified assumptions can be confined to just one or two relevant terms without affecting correctness: *e.g.* instead of a generic

```
(ALL n. n = 0() | (EX m. n = s(m) & ...)) --> ...
```

it might be sufficient to consider the cases for some fixed `n` only

```
ALL n. (n = 0() | (EX m. n = s(m) & ...)) --> ...
```

4.3 Concluding remarks

assertion is a generic first-order logic prover and not specifically tuned for term rewriting. Therefore a tautology whose proof is based on a lengthy chain of equational transformations may pose a challenge concerning computational resources so that providing an intermediate result from the chain in form of an assumption is basically helpful. It is moreover worth trying to regroup terms (as far as it is backed by the given assumptions, *e.g.* associativity): for instance

```
assertion "join-mono1" :
(ALL a. a + a = a) & (ALL a b. a + b = b + a) &
(ALL a b c. a + (b + c) = (a + b) + c) -->
(ALL a b x. a + b = b --> a + (x + (b + x)) = b + x);
```

```
assertion "join-regroup" :
(ALL a. a + a = a) & (ALL a b. a + b = b + a) &
(ALL a b c. a + (b + c) = (a + b) + c) -->
(ALL a b x. a + (x + (b + x)) = (a + x) + (b + x))
```

gets confirmed within seconds whereas the direct

```
assertion "join-mono" :
(ALL a. a + a = a) & (ALL a b. a + b = b + a) &
(ALL a b c. a + (b + c) = (a + b) + c) -->
(ALL a b x. a + b = b --> (a + x) + (b + x) = b + x)
```

may by contrast take minutes. Also flipping the sides of an equality might occasionally have positive effects. Similar applies to function arguments: for instance in the already mentioned group-theoretic setting, the conditions

```
(ALL a. 1() * a = a) & (ALL a. i(a) * a = 1())
```

and

```
(ALL a. a * 1() = a) & (ALL a. a * i(a) = 1())
```

entail each other with an associative `*` and, although both orientations are largely useful, in some settings it might be more beneficial to assume either only the former or the latter one.

Lastly, defining extra relations by means of assumptions often results in more structured formulations but on the other hand may also lead to complications, especially for equational reasoning. Therefore the rule is:

– *if all relations in a formula (except equality, of course) are defined in terms of other symbols without circularities then equational reasoning is more effective when the definitions are unfolded and the respective assumptions discarded.*

So, for instance

```
assertion "unfolded" :
(ALL a. a + a = a) & (ALL a b. a + b = b + a) &
(ALL a b c. a + (b + c) = (a + b) + c) -->
(ALL a b x. a + x = x & b + x = x --> (a + b) + x = x)
```

gets confirmed very quickly whereas

```
assertion "folded" :
(ALL a. a + a = a) & (ALL a b. a + b = b + a) &
(ALL a b c. a + (b + c) = (a + b) + c) &
(ALL a b. a <= b <-> a + b = b) -->
(ALL a b x. a <= x & b <= x --> a + b <= x)
```

with the relation `<=` explicitly defined using an extra assumption, demands incomparably more resources.

The above rule should not be simply generalised though: in cases that are not covered by its premises, an extra definitional assumption does not necessarily pose an obstacle for reasoning, *i.e.* various approaches might be worth trying.