Developing a Loop Invariant in KeY-Hoare: Fibonacci Example

Moa Johansson

December 5, 2012

1 Introduction

Discovering a good loop invariant is an art. For all but the most trivial programs, we often have to develop the invariant in stages, gradually refining it until we have a formula which holds before, during and after the loop has been executed. Tools, such as Key-Hoare can however be used to check candidate invariants and to give us a hint as to how to patch faulty candidate invariant in cases when the proofs do not go through.

This note contains a worked example of how to use KeY-Hoare to help develop a loop invariant for a program which computes Fibonacci numbers, which makes up the familiar sequence $1, 1, 2, 3, 5, 8, 13, \ldots$. We write $fib(n)$ for the n_{th} Fibonacci number. The first two Fibonacci numbers are both 1, so to start off the sequence, we define

$$fib(1) = fib(2) = 1$$

For all Fibonacci numbers greater than 2, the n_{th} Fibonacci number is computed by taking the sum of the two previous Fibonacci numbers:

$$fib(n) = fib(n-1) + fib(n-2)$$

2 Fibonacci in KeY-Hoare

We start by specifying the program using a Hoare triple with updates. Here, `n` is the Fibonacci number we want to compute, and `n0` denote its initial value as usual in symbolic evaluation. We know that the pre-condition at least requires `n` to be greater than 0 (otherwise the n_{th}Fibonacci number is not defined) and that `n` has initial value `n0`.

```
{ n = n0 & n > 0 }

 x1 = 1;
 x2 = 1;
 while (n > 2) {
   x2 = x1 + x2;
   x1 = x2 - x1;
   n = n - 1;
 }

 {???}
```

But how do we specify the postcondition? We would like to refer to the logical function fib, from the Introduction! Luckily, we can simply add this function to our specification:

```
{ n = n0 & n > 0 &
 fib(1) = 1 &
 fib(2) = 1 &
 \forall int m; (m > 2 -> fib(m) = fib(m - 1) + fib(m - 2))
 }

 x1 = 1;
 x2 = 1;
 while (n > 2) {
   x2 = x1 + x2;
   x1 = x2 - x1;
   n = n - 1;
 }

 {x2 = fib(n0)}
```

OK, great, we have managed to specify pre- and postconditions for our program. Now, we want to find a *loop invariant*. This invariant will need to express some complex relation between the loop and the fib function. How do we find it?

2.1 Unwinding the Loop to find Patterns

Let's unwind the loop a couple of iterations to see if we can find some pattern in how the variables `x1, x2, n` are updated:

Iteration	`x1`	`x2`	`n`
0	$fib(1)$	$fib(2)$	`n0`
1	$fib(2)$	$fib(3)$	`n0 -1`
2	$fib(3)$	$fib(4)$	`n0 -2`

If we look at this table, and think hard for a little while, we can conjecture that the values of `x1, x2` seem to follow the pattern:
$\mathtt{x1} = \mathtt{fib}(\mathtt{n0} - \mathtt{n} + 1)$
$\mathtt{x2} = \mathtt{fib}(\mathtt{n0} - \mathtt{n} + 2)$
We will use this in our invariant.

2.2 A first try at an Invariant

Recall the invariant rule (if not, look in the lecture notes), and notice that in the *Preserves case*, where we prove that the invariant holds for each loop iteration, does not include the `fib` function, which was defined in the precondition. Therefore, we need to include the definition of `fib` also in the invariant. So our first attempt at an invariant is:

```
x1 = fib(n0 - n + 1) &
x2 = fib(n0 - n + 2) &
fib(1) = 1 &
fib(2) = 1 &
\forall int m; (m > 2 -> fib(m) = fib(m - 1) + fib(m - 2))
```

Let's check if the post-condition follows from this invariant (The *Use Invariant* case in Key-Hoare). To arrive at the post-condition, the loop guard, `n>2` must be false, in other words we know that `n <=2` (as this is equivalent to `!(n>2)`). Hence we check if the *invariant and the negated loop condition implies the postcondition* `x2 = fib(n0)`:

```
 |-
   x1 = fib(n0 - n + 1)                                                      \\Inv
  & x2 = fib(n0 - n + 2)                                                     \\Inv
  & fib(1) = 1                                                               \\Inv
  & fib(2) = 1                                                               \\Inv
  & \forall int m;  (m >  2 -> fib(m) = fib(m - 1) + fib(m - 2))     \\ Inv
  & !n >  2                                         \\ Negated loop guard
->
  x2 = fib(n0)                                      \\Post-condition
```

If you try to an automated proof in Key-Hoare the Use Invariant case will fail. However, it is true provided that `n >=2`, which it needs to be for the loop to be executed at all!. We refine the invariant to:

```
n >= 2 &
x1 = fib(n0 - n + 1) &
x2 = fib(n0 - n + 2) &
fib(1) = 1 &
fib(2) = 1 &
\forall int m; (m > 2 -> fib(m) = fib(m - 1) + fib(m - 2))
```

2.3 Strengthening the Invariant Further

OK, so let's try this new invariant in Key-Hoare. Remember, right click on the highlighted goal, and choose Loop Invariant from the menu (see Figure 1).

Without spoiling the surprise, KeY-Hoare won't be able to prove the three cases automatically either. We will need further refinements. To get a hint as to why, let's look at the Preserves Invariant Case in Figure 2.

You should know the drill by now, we right-click and choose the Assignment rule to symbolically execute the three statements from the body of the while loop. Then, we apply the exit rule.

To apply the updates to the goal, we next choose Update Simplification from the menu. Now we would probably try to discharge the proof automatically. Try it! Unfortunately, the automated prover fails to prove the goal, and just produce a long sequence of gibberish. Let's backtrack, by clicking on the last one of our goals (Update Simplification) and the clicking the Goal Back button at the top. The current state is shown in Figure 3.

So let's try to get some hints as to what part the prover failed on, so we can use this information to refine the invariant. We will do this by stepping through the proof interactively to see exactly how far we get, and exactly which subgoal that fails. As the goal in Figure 3 is a conjunction we need to prove that *each of the conjuncts* is true. To do this in KeY-Hoare, we want to split the goal into one subgoal for each conjunct, and prove each one separately. Key-Hoare has a rule *andRight*[1] which does just this. As usual, right click on the goal, (more specifically, on the last `&`-sign in the goal) and choose the rule from the menu. This helps us in pinpointing more precisely in which sub-goal (conjunct) things fail in the proof.

After having applies the *andRight* rule once for each one of the conjuncts, we arrive at the state with six open subgoals in Figure 4.

All but the second of these subgoals can be solved automatically (recall, right-click on the OPEN GOAL in the Proof panel, and choose *Apply Strategy*). Remember that you can backtrack by clicking the *Goal Back* button on goals that the prover don't solve automatically (as it is usually getting itself lost trying to apply all sorts of strange rules). Instead, we step through this goal interactively. As shown in Figure 5, we have applied the rule `andLeft`, to the premises, which takes apart the large conjunction (essentially removing brackets, which confusingly are not displayed by KeY-Hoare anyway...). Having done this allows us to use these equations individually,

[1]It is called *andRight* as we are splitting a conjunction which occurs on the right hand side (conclusion) of the implication arrow. The corresponding rule for the left hand side (premises) is called *andLeft*.

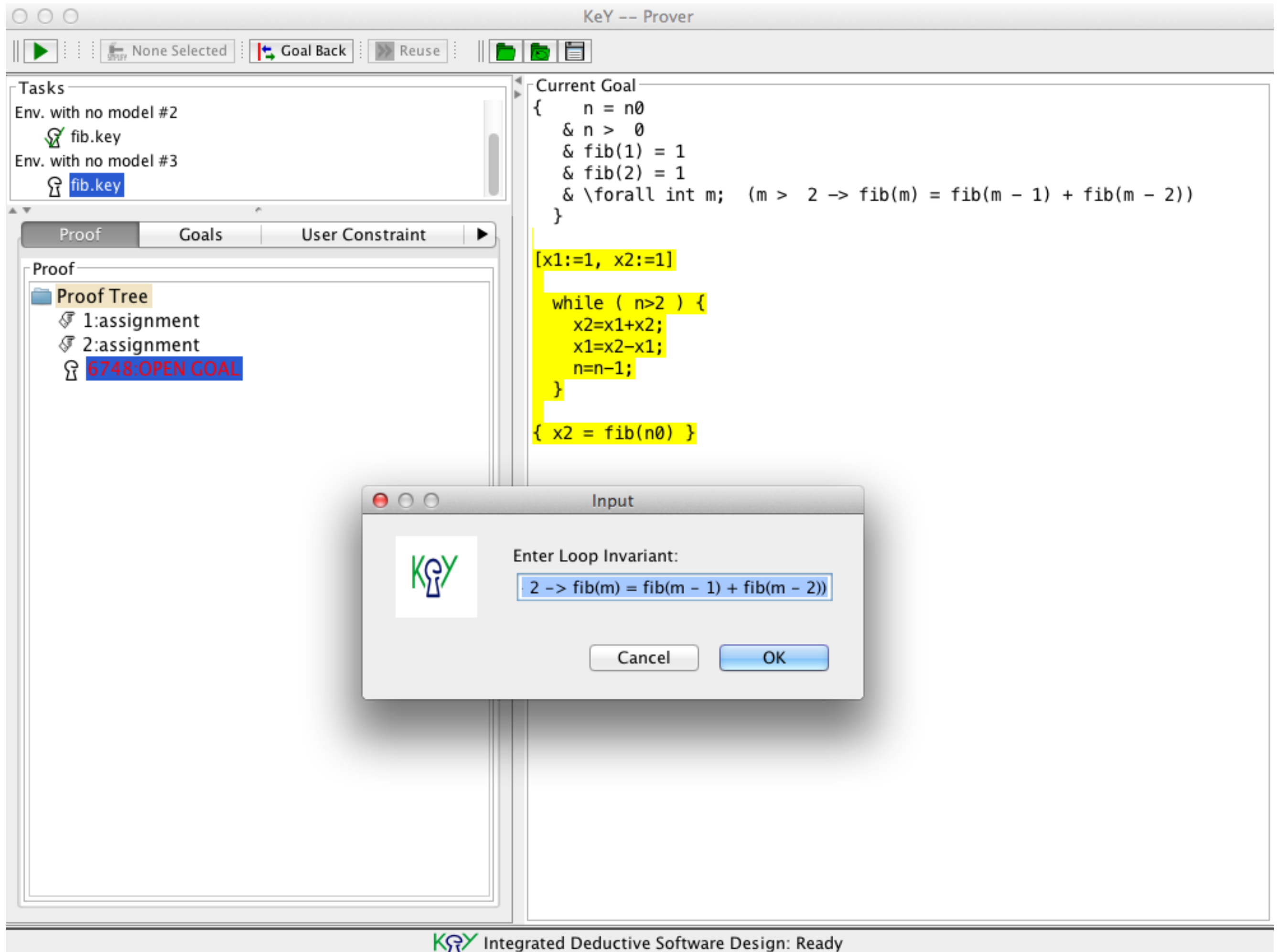

Figure 1: Inserting a Loop Invariant

and thus replace the variables `x1` and `x2` in the goal by their respective values `fib(n0 - n + 1)` and `fib(n0 - n + 2)`. To do this in KeY-Hoare, right click on the relevant variable and choose *Apply Equation*. Having applied these equations we have:

```
fib(n0 - n + 1) + fib(n0 - n + 2)) = fib(n0 - (n - 1) + 2
```

Let's stop Key-Hoare there and think a little bit about our invariant again. To make things a little easier to read, the last open sub-goal is equivalent to:

```
fib(n0 - n + 3) = fib(n0 - n + 1) + fib(n0 - n + 2)
```

This looks alright, we can simply plug in `n0 - n + 3` in place of the argument `m` in the definition of `fib` and we should be done:

$$\forall \; \texttt{int m}; \; (\texttt{m} > 2 \rightarrow \texttt{fib(m)} = \texttt{fib(m} - 1) + \texttt{fib(m} - 2))$$

Ah! This requires `n0 - n + 3` $>$ 2, which is equivalent to `n0 > n -1`. But, as the pre-condition state that `n = n0` in the beginning, let's try adding `n0 >= n` to the invariant. The strengthened invariant candidate is now:

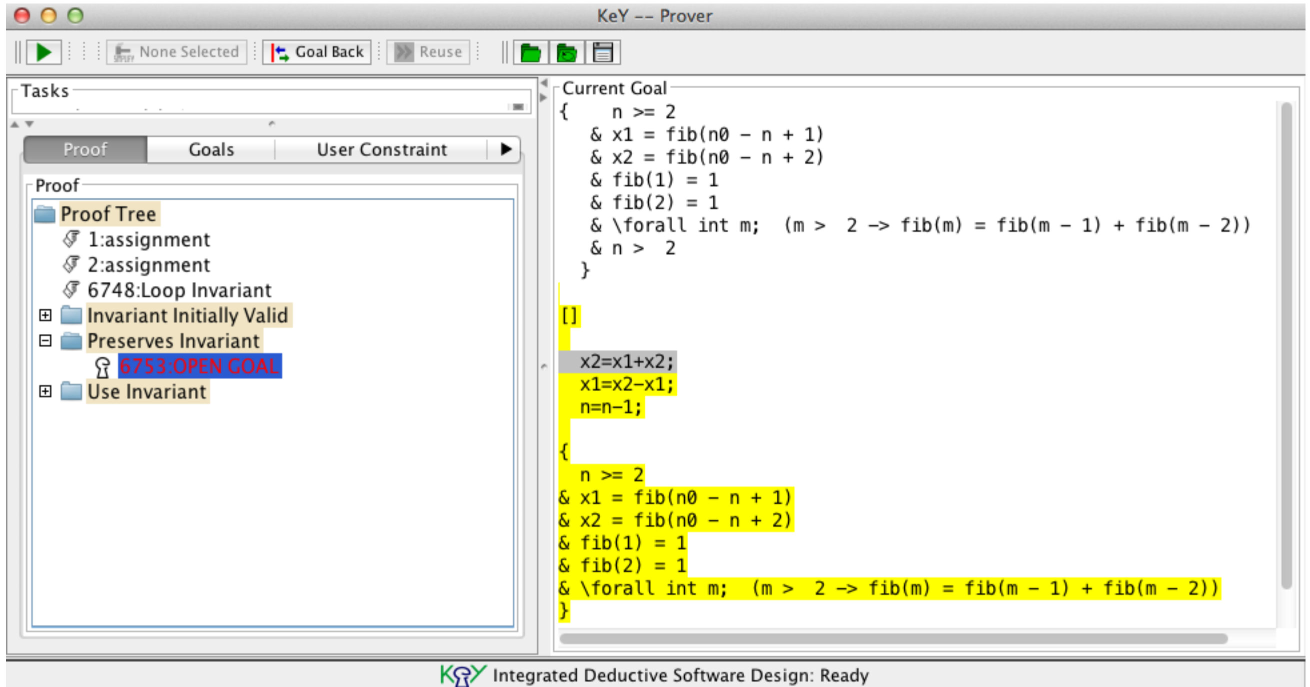

Figure 2: The Preserves Invariant case.

```
n >= 2 &
   n0 >= n &
   x1 = fib(n0 - n + 1) &
   x2 = fib(n0 - n + 2) &
   fib(1) = 1 &
   fib(2) = 1 &
   \forall int m; (m > 2 -> fib(m) = fib(m - 1) + fib(m - 2))
```

2.4 The Final Invariant

Using the invariant candidate from the previous section we are almost there! Now, both the *Preserves Invariant* and *Use Invariant* cases go through! However, we need to have a look at the *Initially Valid* case. Splitting the conjunction using the *andRight* rule as before gives us a number of subgoals. All but the first can be proved automatically. The remaining goal is shown in Figure 6. Clearly, we cannot show that

```
|-
    n = n0
  & n >  0
  & fib(1) = 1
  & fib(2) = 1
  & \forall int m;  (m >  2 -> fib(m) = fib(m - 1) + fib(m - 2))
->
  n >= 2
```

The conjunct `n>=2` from the invariant is too strong. Recall that we added this conjunct at an early stage in order to get an invariant candidate which implied the post-condition after the loop had been executed (see §2.2). But to prove the above, there seems to be something missing for

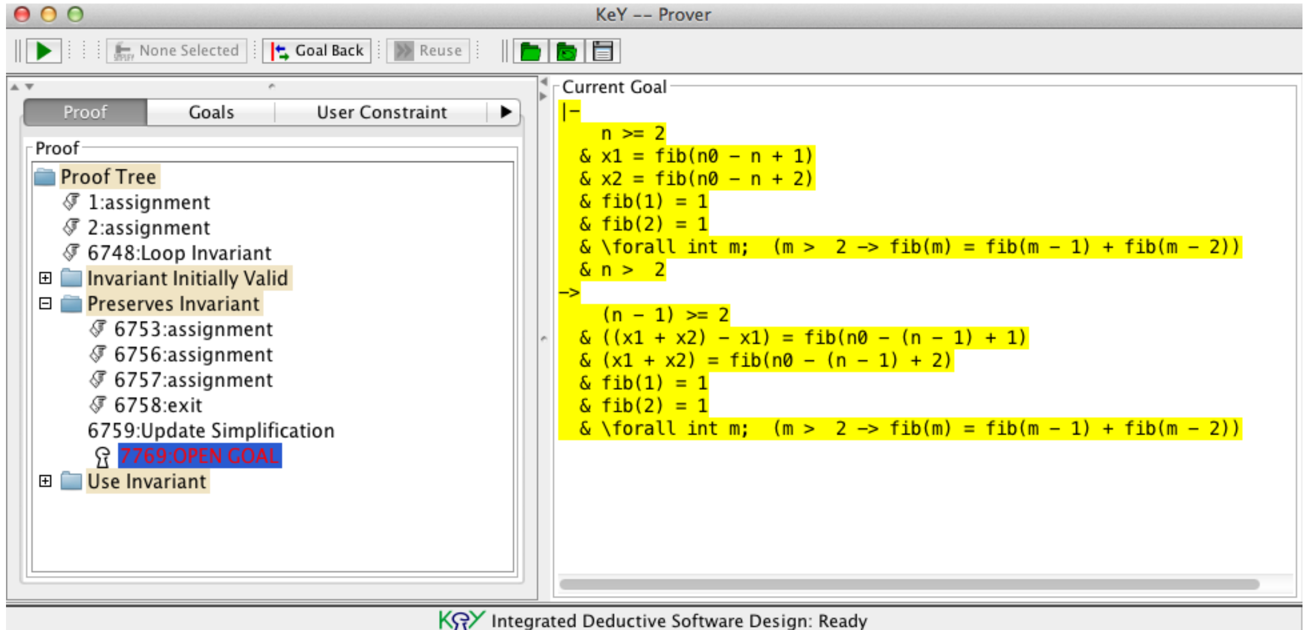

Figure 3: The Preserves Invariant case, after having symbolically executed the loop body. We need to show that the Invariant and Loop Guard implies the Invariant after the updates occurring in the loop body.

when `n=1`, which is a valid input, the precondition only requires that `n > 0`. Suppose we *weaken* the invariant, by replacing the conjunct `n >= 2` by `n > 0` from the precondition.

The candidate invariant is then

```
n > 0 &
   n0 >= n &
   x1 = fib(n0 - n + 1) &
   x2 = fib(n0 - n + 2) &
   fib(1) = 1 &
   fib(2) = 1 &
   \forall int m; (m > 2 -> fib(m) = fib(m - 1) + fib(m - 2))
```

This candidate is indeed initially valid, and preserved by the loop body, but it does not imply the post-condition after to execution of the loop! Let's analyse this subgoal (shown in full in Figure 7).

```
|-
    n >  0
  & n0 >= n
  & x1 = fib(n0 - n + 1)
  & x2 = fib(n0 - n + 2)
  ... def of fib omitted ...
  & !n >  2
->
  x2 = fib(n0)
```

Observe that we can apply the equation specifying `x2 = fib(n0 - n +2)` to the conclusion to obtain

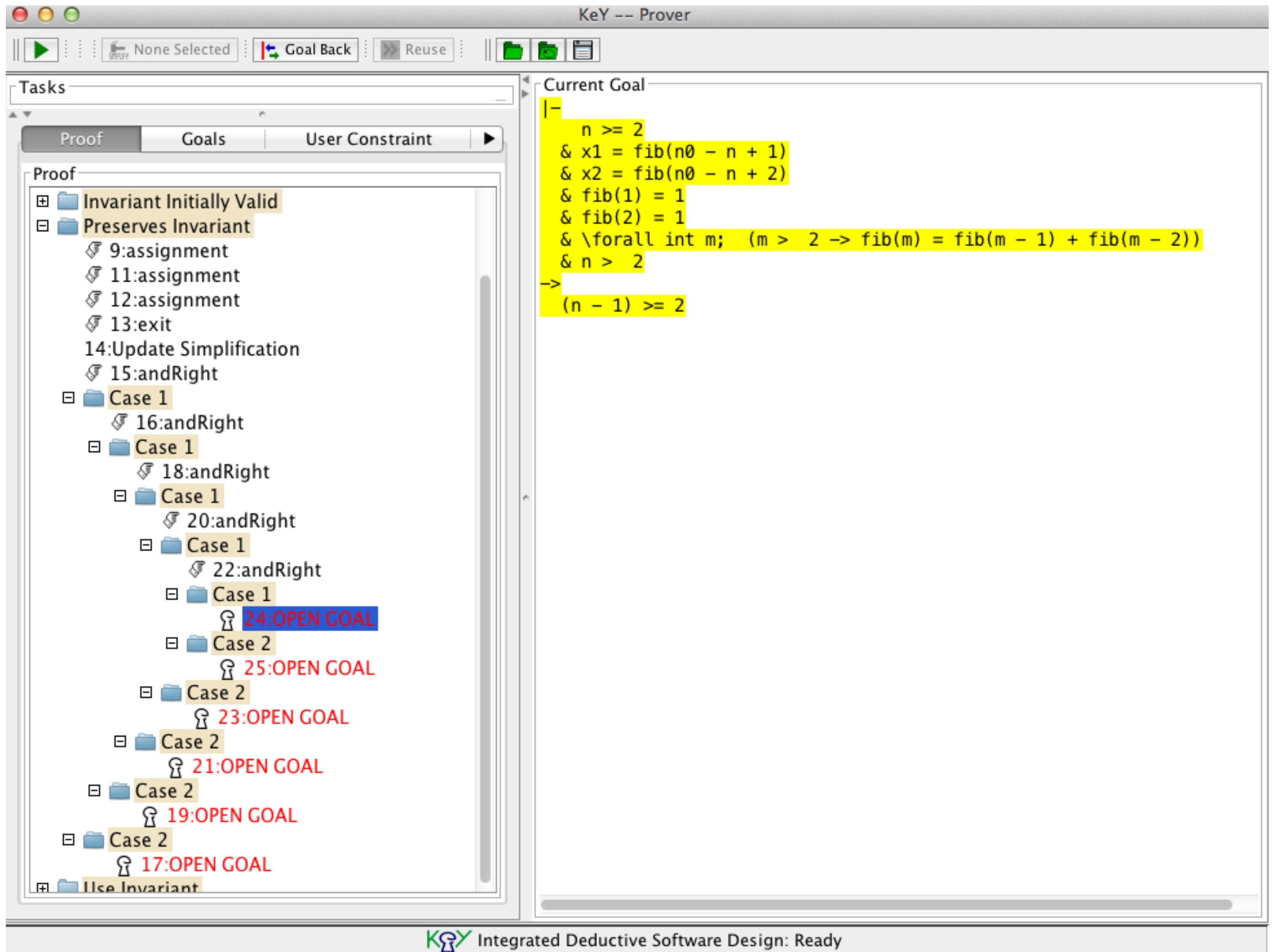

Figure 4: Splitting the goal into six smaller sub-goals, by applying the *andRight* rule.

```
|-
...
->
  fib(n0 - n + 2) = fib(n0)
```

Furthermore, in the premises we have two conjuncts `n > 0` and `!n > 2`, which together imply that `n=2` or `n=1` at the end of the loop. Let's substituting these values for `n` in the above. When `n=2` we get: `fib(n0) = fib(n0)`, which is trivially true, so the problem does not arise from this case. When `n=1` we get: `fib(n0 + 1) = fib(n0)`. From the definition of `fib`, we know that it is just the first two Fibonacci numbers which are equal, thus `n0` must be `1` whenever `n =1`! Adding the conjunct `n=1 -> n0 =1` to the invariant gives us the final version:

```
n > 0 &
    (n = 1 -> n0 = 1) &
    n <= n0 &
    x1 = fib(n0 - n + 1) &
    x2 = fib(n0 - n + 2) &
    fib(1) = 1 &
    fib(2) = 1 &
    \forall int m; (m > 2 -> fib(m) = fib(m - 1) + fib(m - 2))
```

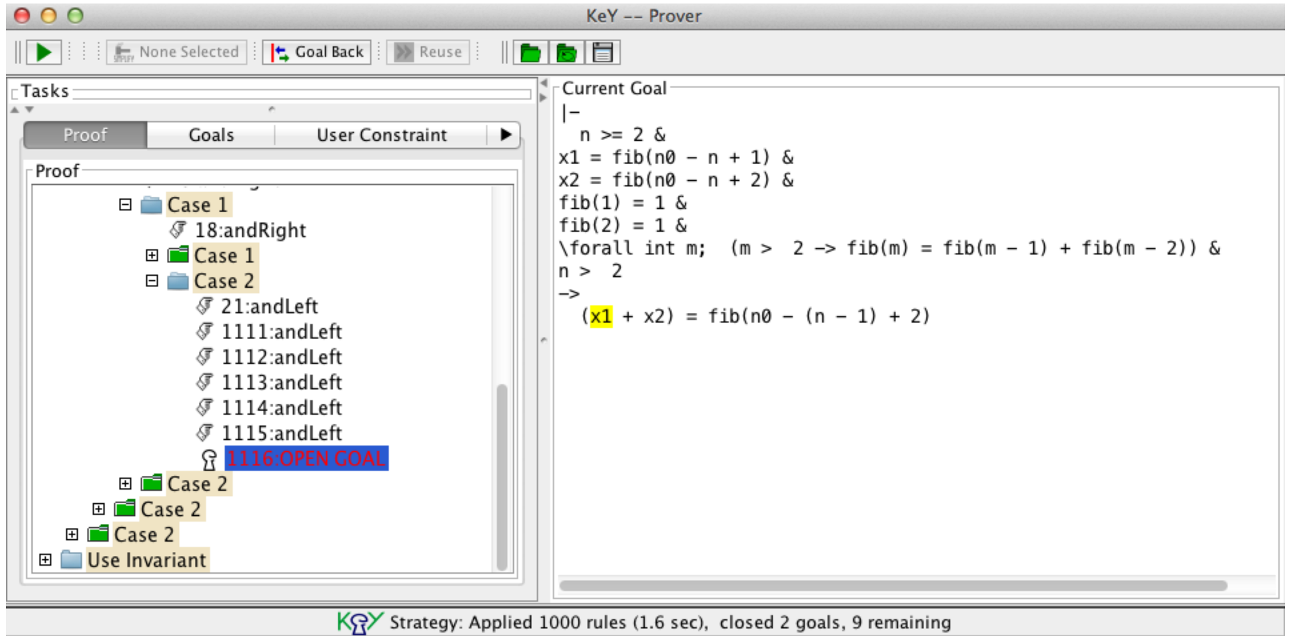

Figure 5: Stepping through the second sub-goal, by applying the *andLeft* rule, taking the big conjunction in the premises apart, so we can use each one individually.

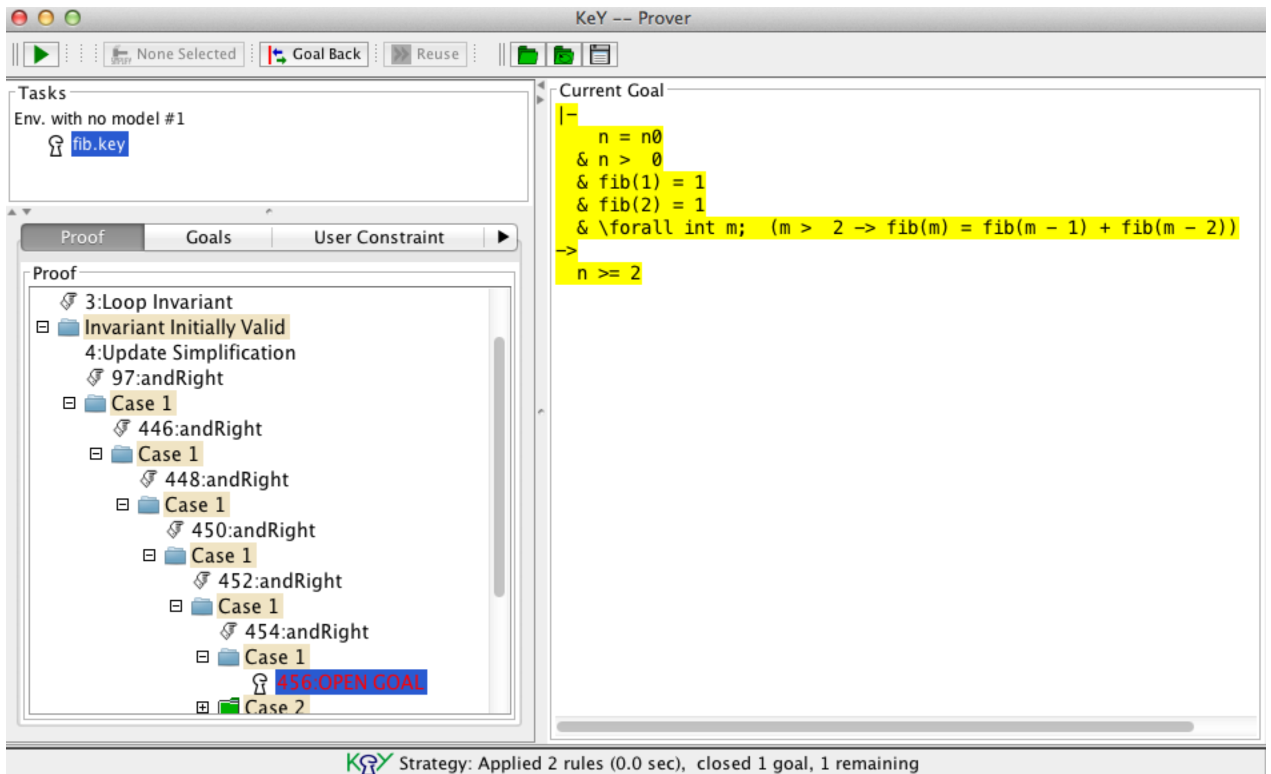

Figure 6: This time, there is a problematic subgoal in the *Invariant Initially Valid* case, as the conjunct `n >= 2` from the invariant is too strong.

Try this invariant in Key-Hoare and watch it prove all three cases automatically!

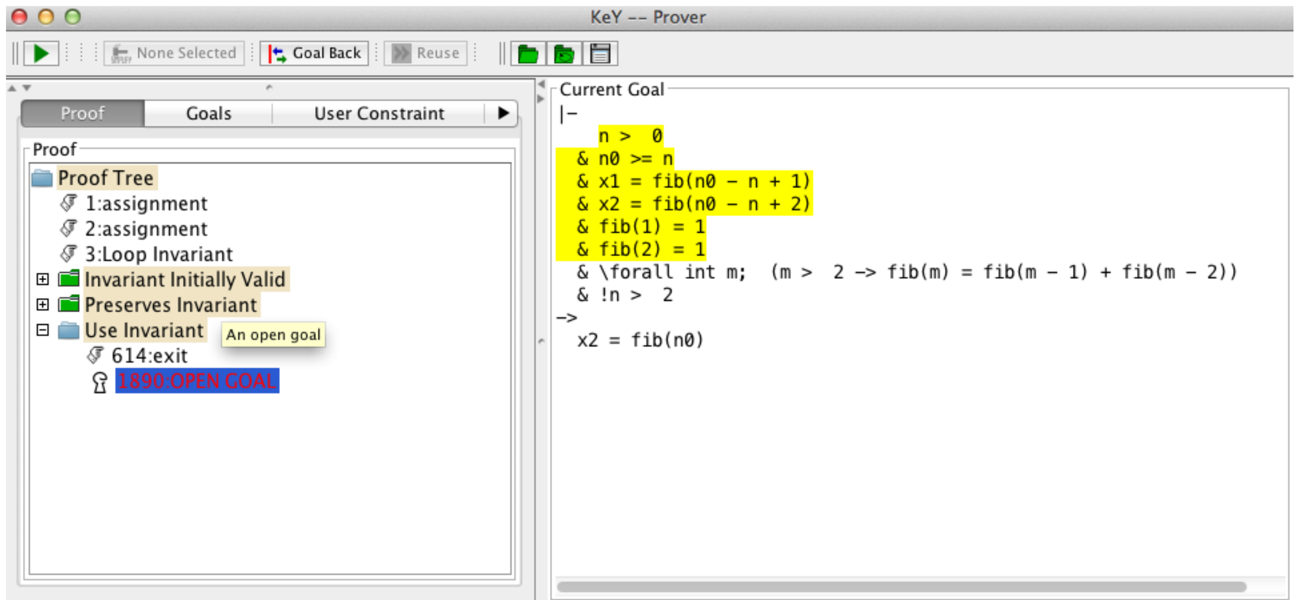

Figure 7: We have weakened the invariant. The *Use Invariant* case does still not go through.