ProVerif is well documented, see https://bblanche.gitlabpages.inria.fr/proverif/. These notes are just a small complement to highlight a few facts that may come in handy during the lab.

ProVerif represents data as terms in a term algebra. Terms are constants, variables or constructors applied to other terms. Constructors can be nested. For example,

```
type int.
free u, v: int.
```

defines two constants of a newly introduced type `int`. They are public and known to all processes, including the attacker. We can declare the constants as private. They are then not known to the attacker to begin with.

```
free u, v: int [private].
```

Constructors are defined using the keyword `fun`, e.g., as follows

```
fun f(int): int.
fun g(int, int): int.
```

The constructor `f` takes a term of type `int` and produces another term, also of type `int`. The constructor `g` takes two terms of type `int` and produces a term of type `int`. Constructors are syntactic constructs, sometimes referred to as *uninterpreted functions*. They are not functions, but can be thought of as such for the purpose of ProVerif modeling. Using `f` and `g`, and the two constants `u` and `v`, we can construct terms such as `f(u)` and `f(g(u, f(v)))`.

Constructed terms can be deconstructed using equations or `reduc`. See the manual for differences between the two. For the lab we only need to know that terms can be deconstructed. Consider the following functions and equation, which use the type `int`.

```
fun f(int): int.
fun finv(int): int.
equation forall x: int; finv(f(x)) = x.
```

The equation relates terms built using the constructors `f` and the suggestively named constructor `finv`. The equation lets ProVerif know that if a term of the shape `finv(f(x))` appears, regardless of what term `x` is, `finv(f(x))` can be replaced by the term `x` itself. Note that `finv` is not a function computing inverses. If it were, we could write `f(finv(x))`, and expect the equation reducing that term to `x`. Given the equation above, the term `f(finv(x))` is, however, irreducible.

As discussed in the lecture, ProVerif maintains the attackers' knowledge as

terms. Attackers' knowledge is expanded when they observe a protocol message (modelled as a term) being sent over a public channel. Once attackers learn a new term, they immediately (and without conceptual computational cost!) expand their knowledge by

- deconstructing the term using all equations, `reduc`s and all terms they already know,
- constructing all terms possible from the so obtained terms and existing constructors.

These two steps are recursively applied until every term in the knowledge has been maximally deconstructed into minimal irreducible terms, and all terms possible to construct from those minimal terms, have been constructed.

An important aspect for modeling is that attackers only can construct and deconstruct terms when when they know all subterms of the outermost constructors. To give a concrete example, suppose we have the following functions and equation:

1. `fun f(int): int.`
2. `fun finv(int): int.`
3. `fun g(int, int): int.`
4. `fun ginv(int, int): int.`
5. `equation forall x: int; finv(f(x)) = x.`
6. `equation forall x: int, y: int; ginv(x, g(x, y)) = x.`

Suppose a message, represented by a term $m =$ `f(g(u, g(u, v)))`, is sent on a public channel. Also suppose the attackers' knowledge is {`u`}.

Using 5 and 2 the attackers derive `g(v, g(u, u))` and adds it to their knowledge. Attackers cannot deconstruct `g(v, g(u, u))` further because they don't know `v`. The attacker knowledge is now {`u`, `g(v, g(u, u))`}.

The attackers now construct all terms possible from the knowledge and constructors 1, 2, 3 and 4. There are infinitely many of those, e.g., `f(u)`, `f(f(u))`, `f(f(f(u)))`, `g(u, u)`, and `g(f(u), finv(u))`.

While the attackers do not know the term `v`, they can still construct terms using `g(v, g(u, u))` obtained from the message m. For example, by applying 2 to it and obtaining `finv(g(v, g(u, u)))`.