# Computer-Aided Modelling and Reasoning Project Description (AS 2015)

Andreas Lochbihler and Christoph Sprenger

November 2, 2015

**Abstract**

In this project, we model and prove the correctness of a constraint-based security protocol analyzer. The project consists of two parts. In the first part, we formalize a unification algorithm for general first-order terms and show its soundness, completeness, and termination. In the second part, we specialize this theory to protocol messages and formalize constraint systems and a reduction relation, which we show to be sound and terminating. Finally, we implement a functional program acting as a constraint solver and show its soundness and completeness with respect to the reduction relation.

## 1 Introduction

This project consists of two parts. In the first part, we model general first-order term algebras, equations, and equation solving (unification). Unification is a fundamental primitive at the heart of automated reasoning. For example, the application of a lemma to a proof goal requires the unification of that goal with the lemma's conclusion. Here, we will formalize a unification algorithm for first-order terms and prove its soundness and completeness.

In the second part, we build upon the first part to formalize a constraint solver for security protocol analysis. Security protocols are small distributed programs that use cryptography to achieve a security goal such as entity authentication or session key establishment. Typical examples are SSL/TLS, IPsec, and Kerberos. Notwithstanding their limited size, they can exhibit subtle design flaws that give rise to attacks. Formal analysis of security protocols can find (or exclude) such attacks. We will formalize and implement a constraint solver for security protocol analysis and prove its correctness.

These notes describe the theory behind the project and define the assignments. The presentation of the theory is intentionally kept in a standard mathematical style. We consider it part of the formalization work to translate such a presentation into suitable structures in the theorem prover Isabelle. The assignments in Sections 2.4 and 3.3 provide numerous hints to help you in making the design decisions during the formalization process. Moreover, we have intended to provide sufficient details in our informal proofs to guide the formal ones. Last but not least, we strongly recommend that you ask us questions in order to swiftly resolve issues that pop up during the project.

## 2 Unification of First-order Terms

A first-order signature consists of a set of symbols $\Sigma$ together with an arity function $ar : \Sigma \to \mathbb{N}$. Let $\mathcal{V}$ be a set of variables. The set of first-order terms generated by the signature $\Sigma$, denoted by $\mathcal{T}_\Sigma(\mathcal{V})$, is the least set such that

(i) $\mathcal{V}$ is contained in $\mathcal{T}_\Sigma(\mathcal{V})$, and

(ii) for all $f \in \Sigma$, if $t_1, \ldots, t_{ar(f)} \in \mathcal{T}_\Sigma(\mathcal{V})$ then $f(t_1, \ldots, t_{ar(f)}) \in \mathcal{T}_\Sigma(\mathcal{V})$.

Function symbols of arity 0 are called *constants*. The set $\mathcal{T}_\Sigma(\mathcal{V})$ is also called the *free term algebra* over the signature $\Sigma$. We denote by $fv(t)$ the set of (free) variables in a term $t$. We also define the size of a term $t$, denoted by $|t|$, by $|x| = 0$ and $|f(t_1, \ldots, t_n)| = 1 + \Sigma_{i=1}^n |t_i|$.

By convention, we use $x, y, z$ to denote variables, $a, b, c$ for constants, $f, g, h$ for function symbols of arity at least one, and $t$, $u$, $v$ for arbitrary terms.

### 2.1 Substitutions

A substitution is a function $\sigma : \mathcal{V} \to \mathcal{T}_\Sigma(\mathcal{V})$ that maps each variable to a term. We homomorphically lift substitutions to all terms by defining $\sigma \cdot t$, the application of the substitution $\sigma$ to the term $t$ by

$$\begin{array}{rcl} \sigma \cdot x & = & \sigma(x) \\ \sigma \cdot f(t_1, \ldots, t_{ar(f)}) & = & f(\sigma \cdot t_1, \ldots, \sigma \cdot t_{ar(f)}) \end{array}$$

Given this lifting, we can compose substitutions by defining

$$(\sigma \circ_s \tau)(x) = \sigma \cdot \tau(x).$$

We can then prove the property $(\sigma \circ_s \tau) \cdot t = \sigma \cdot (\tau \cdot t)$ as well as the associativity of composition and that the identity substitution *id* indeed behaves as the identity on both sides of a composition.

The *domain* of a substitution $\sigma$ is the set of variables that do not map to themselves, i.e., $dom(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$. The *range* of a substitution $\sigma$ is the image of its domain under $\sigma$, i.e., $ran(\sigma) = \{\sigma(x) \mid x \in dom(\sigma)\}$. The *variable range* of a substitution $\sigma$ is the set of free variables of terms in its range, i.e., $vran(\sigma) = \bigcup_{t \in ran(\sigma)} fv(t)$.

We write substitutions with a finite domain as $\sigma = [x_1 := t_1, \ldots, x_n := t_n]$. Here, we have $dom(\sigma) = \{x_1, \ldots, x_n\}$ and $ran(\sigma) = \{t_1, \ldots, t_n\}$ (assuming that $x_i \neq t_i$ for all $i$ such that $1 \leq i \leq n$).

**Example 1.** $[x := y, y := t] \cdot f(x, g(y)) = f(y, g(t))$.

The following lemma gives a useful upper bound on the free variables of a term after substitution.

**Lemma 1.** *$fv(\sigma \cdot t) \subseteq (fv(t) \setminus dom(\sigma)) \cup vran(\sigma)$.*

## 2.2 The unification problem

A *unification problem* $U$ over $\Sigma$ is a finite set of *equations* over terms in $\mathcal{T}_\Sigma(\mathcal{V})$:

$$U = \{(t_1, u_1), \ldots, (t_n, u_n)\}$$

A *unifier* for $U$ is a substitution $\sigma$ such that $\sigma \cdot t = \sigma \cdot u$ for all $(t, u) \in U$, i.e., $\sigma$ solves the equation system defined by $U$.

A substitution $\sigma$ is the *most general unifier* (mgu) for $U$ if it is a unifier for $U$ and any other unifier $\tau$ for $U$ factors through $\sigma$, i.e., $\tau = \rho \circ_s \sigma$ for some substitution $\rho$. Note that the mgu is unique only up to variable renaming. For example, $[y := x]$ and $[x := y]$ are both mgus of $U = \{(x, y)\}$. They factor through each other by the renaming $[x := y, y := x]$.

**Example 2.** Let $t = f(x, g(y))$ and $u = f(h(a, z), z)$. The unification problem $U = \{(t, u)\}$ has the most general unifier $\sigma = [x := h(a, g(y)), z := g(y)]$. The substitution $\tau = [x := h(a, g(t)), y := t, z := g(t)]$ for any term $t$ is also a unifier for $U$, but a less general one: $\tau = [y := t] \circ_s \sigma$.

## 2.3 Robinson-style unification algorithm

We give a pseudo-code specification of the unification algorithm (see Algorithm 1). The algorithm takes a unification problem $U$ as argument and either (i) succeeds and returns a substitution $\sigma$ or (ii) fails and returns $\perp$. If there are no equations left in $U$ we return the trivial unifier (line 3). Otherwise, we pick an equation $(u, t) \in U$ to process (line 5). We denote by $U'$ the set $U$ with the equation $(u, t)$ removed. We distinguish four cases. We label all cases with symbolic names that we can later refer to in our proofs.

**Var** $u$ is a variable $x$. We further distinguish three subcases.

- **Unify** If $x$ does not appear free in $t$ then we compose the substitution $\sigma = \text{UNIFY}([x := t] \cdot U')$ obtained by the recursive call with $[x := t]$, i.e., we return $\sigma \circ_s^\perp [x := t]$. Here, $\sigma \circ_s^\perp \tau$ is the lifted composition which equals $\perp$ if its first arguments is $\perp$ and $\sigma \circ_s \tau$ otherwise.
- **Simp** In the trivial case where $x = t$ we simply recurse on $U'$.
- **Occur** We have $x \notin fv(t)$ and $x \neq t$ meaning that there is no unifier for $x$ and $t$ and we return $\perp$. This is the so-called *occurs-check*.

**Swap** The term $t$ is a variable (but $u$ is not). In this case, we recurse on $U' \cup \{(t, u)\}$, replacing the equation $(u, t)$ by its symmetric version $(t, u)$.

**Algorithm 1** Unification algorithm in pseudo-code

```
function UNIFY(U)                                         {$U$ is a unification problem}
    if $U = \emptyset$ then
        $id$                                                       {trivial unifier}
    else
        let $(u,t) \in U$ and $U' = U \setminus \{(u,t)\}$ in                 {pick an equation}
        if $u$ is a variable $x$ then
            if $x \notin fv(t)$ then
                UNIFY($[x := t] \cdot U'$) $\circ_s^{\perp}$ $[x := t]$        {Unify: extend unifier}
            else if $x = t$ then
                UNIFY($U'$)                           {Simp: remove trivial equation}
            else
                $\perp$                                  {Occurs: $x \in fv(t)$ and $x \neq t$}
            end if
        else if $t$ is a variable $x$ then
            UNIFY($U' \cup \{(t,u)\}$)                        {Swap: reorient equation}
        else if $u = f(u_1, \ldots, u_n)$ and $t = f(t_1, \ldots, t_n)$ then
            UNIFY($U' \cup \{(u_i,t_i) \mid 1 \leq u \leq n\}$)      {Fun: decompose terms}
        else
            $\perp$                                {Fail: different function symbols}
        end if
    end if
end function
```

**Fun** The terms $u$ and $t$ have the same top-level function symbol $f$. Here, we call UNIFY recursively, replacing the original equation by the (possibly empty) set of pairs of arguments $(u_i, t_i)$ of $u$ and $t$.

**Fail** The terms $u$ and $t$ are built using different function symbols. Here, there is no unifier and we therefore return $\perp$.

The main results regarding this algorithm are its soundness, completeness, and termination. Together they state that the algorithm returns a most general unifier if it exists and fails (i.e., returns $\perp$) otherwise. We start by stating and proving the termination property as this will allow us to prove soundness and completeness by induction on the definition of UNIFY (computation induction).

**Theorem 1** (Termination of UNIFY)**.** *Algorithm 1 terminates on all unification problems $U$.*

*Proof.* We prove termination by showing that the lexicographic combination of three measure functions decreases with every recursive call. We consider the cases **Unify**, **Simp**, **Swap**, and **Fun**. The other two cases, **Occur** and **Fail**, fail and therefore obviously terminate. We use the following three measures on unification problems $U$:

$$\begin{aligned}
\chi_1(U) &= |fv(U)| \\
\chi_2(U) &= \Sigma_{(t,u)\in U}|t| \\
\chi_3(U) &= |U|
\end{aligned}$$

The first one, $\chi_1$, denotes the number of variables in $U$. Clearly, the case **Unify** decreases $\chi_1$ while the other relevant cases do not increase it. The second measure, $\chi_2$, denotes the sum of the sizes of the left-hand-side terms of the equations in $U$. It is decreased by the cases **Swap** and **Fun** and preserved by **Simp**. Finally, rule **Simp** decreases $\chi_3$, the number of equations in $U$.

| | $\chi_1(U)$ | $\chi_2(U)$ | $\chi_3(U)$ |
|---|---|---|---|
| **Unify** | $<$ | | |
| **Simp** | $\leq$ | $=$ | $<$ |
| **Swap** | $=$ | $<$ | |
| **Fun** | $=$ | $<$ | |

These observations are summarized in the table above and conclude the termination proof. □

**Theorem 2** (Soundness of UNIFY)**.** *If* UNIFY *terminates with a substitution $\sigma$ on input $U$ then $\sigma$ is a most general unifier for $U$.*

*Proof.* By induction on the definition of UNIFY (computation induction). In the base case, we have $U = \emptyset$ and $\sigma = id$. Therefore, the statement trivially holds. The inductive steps correspond to the cases **Unify**, **Simp**, **Swap**, and **Fun**.

In the following, we detail the case **Unify** and leave the (easier) remaining cases to the reader. Let $U' = U \setminus \{(x,t)\}$ and $\sigma = \text{UNIFY}([x := t] \cdot U')$. We have to show that $\sigma' = \sigma \circ_s^{\perp} [x := t]$ is an mgu for $U$. By the assumption that UNIFY succeeds on $U$ we have $\sigma \neq \perp$. Hence, we can use $\circ_s$ instead of $\circ_s^{\perp}$ in the following. Since $x \notin fv(t)$, we have

$$\sigma' \cdot x = (\sigma \circ_s [x := t]) \cdot x = \sigma \cdot t = (\sigma \circ_s [x := t]) \cdot t = \sigma' \cdot t.$$

By induction hypothesis, $\sigma$ is an mgu for the unification problem $[x := t] \cdot U'$. It follows that $\sigma'$ is a unifier of $U'$. Hence, $\sigma'$ is a unifier of $U$. It remains to show that $\sigma'$ is the most general unifier.

Suppose $\tau$ is a unifier for $U$. Then $\tau \cdot x = \tau \cdot t$ and $\tau \cdot u = \tau \cdot w$ for all $(u,w) \in U'$. Since $x \notin fv(t)$, we have $\tau \circ_s [x := t] = \tau$. Therefore, $\tau$ is also a unifier for $[x := t] \cdot U'$. By induction hypothesis, $\sigma$ is the mgu for $[x := t] \cdot U'$. Hence, there is a substitution $\rho$ such that $\tau = \rho \circ_s \sigma$. Therefore, we also have $\tau \circ_s [x := t] = \rho \circ_s \sigma \circ_s [x := t]$. Finally, since $\tau \circ_s [x := t] = \tau$, we have $\tau = \rho \circ_s \sigma'$, establishing that $\sigma'$ is the most general unifier of $U$ as required. □

We now turn our attention to completeness. We first establish a lemma stating that UNIFY does not fail if there exists a unifier.

**Lemma 2.** *If there is a unifier for $U$ then* UNIFY *does not fail (i.e., return $\perp$).*

*Proof.* By induction on the definition of UNIFY (computation induction). The statement trivially holds for the base case, since $\text{UNIFY}(\emptyset) = id$.

For the inductive steps, we distinguish the six cases **Unify**, **Simp**, **Occurs**, **Swap**, **Fun**, and **Fail**. In each case, we have to show that the result is not $\perp$ assuming there is a unifier $\tau$ for $U$. Here, we only treat the cases **Unify** and **Occurs**, and leave the remaining cases to the reader.

For **Unify**, suppose $\tau$ unifies $U$. Hence, $\tau \cdot x = \tau \cdot t$. Since $x \notin fv(t)$ we have $\tau \circ_s [x := t] = \tau$. Hence, $\tau \circ_s [x := t]$ unifies $U'$ and thus $\tau$ unifies $[x := t] \cdot U'$. By induction hypothesis, we obtain $\text{UNIFY}([x := t] \cdot U') \neq \perp$. Thus, $\text{UNIFY}([x := t] \cdot U') \circ_s^{\perp} [x := t] = \text{UNIFY}(U) \neq \perp$ as required.

Next, we show that the case **Occurs** is not possible. Suppose that $x \neq t$ and $\tau$ unifies $U$, i.e., $\tau \cdot x = \tau \cdot t$ in particular. As $x \in fv(t)$, we have that $\tau \cdot x$ is a subterm of $\tau \cdot t$, i.e., $\tau \cdot t = f_1(\ldots, f_2(\ldots f_n(\tau \cdot x) \ldots), \ldots)$ for some $n \geq 0$ and $f_1, \ldots, f_n$. Consider the number of function symbols in $\tau \cdot t$ and $\tau \cdot x$. Clearly, the two numbers can be only equal if $n = 0$, i.e., $x = t$, which contradicts the assumption $x \neq t$. □

**Theorem 3** (Completeness of UNIFY)**.** *If there is a unifier for $U$ then* $\text{UNIFY}(U)$ *returns a unifier for $U$.*

*Proof.* Follows directly from Lemma 2 and soundness (Theorem 2). □

We summarize some additional properties of the unification algorithm UNIFY in the following lemma, which we will need later. Note that property (iv) of this lemma is equivalent to stating that $\sigma$ is idempotent, i.e., $\sigma = \sigma \circ_s \sigma$.

**Lemma 3.** *Suppose* $\text{UNIFY}(U) = \sigma$. *Then we have (i) $fv(\sigma \cdot U) \subseteq fv(U)$, (ii) $dom(\sigma) \subseteq fv(U)$, (iii) $vran(\sigma) \subseteq fv(U)$, and (iv) $dom(\sigma) \cap vran(\sigma) = \emptyset$.*

*Proof.* By computation induction on the definition of UNIFY. We first prove points (i)-(iii) together and then point (iv) separately below.

For (i)-(iii), we only consider the most interesting case where $(x, t) \in U$ for a variable $x \in \mathcal{V}$ and leave the other cases to the reader. Let $U' = U \setminus \{(x, t)\}$. We consider the three subcases from lines 7-13 of Algorithm 1.

- Case **Unify**, i.e., $x \notin fv(t)$. The induction hypothesis for this case states that the lemma holds for $[x := t] \cdot U'$ and any $\sigma'$. From the assumption that $\text{UNIFY}(U) = \sigma$ we obtain a $\sigma'$ such that $\text{UNIFY}([x := t] \cdot U) = \sigma'$ and $\sigma = \sigma' \circ_s [x := t]$. We also have $fv([x := t] \cdot U') \subseteq fv(t) \cup fv(U')$. Therefore, we derive from the induction hypothesis that

$$\begin{aligned} fv(\sigma' \cdot ([x := t] \cdot U')) &\subseteq fv(t) \cup fv(U') && (1) \\ dom(\sigma') &\subseteq fv(t) \cup fv(U') && (2) \\ vran(\sigma') &\subseteq fv(t) \cup fv(U') && (3) \end{aligned}$$

For (i), we have to show that $fv(\sigma \cdot U) \subseteq fv(U)$. First, we observe that $fv(U) = fv(t) \cup fv(U') \cup \{x\}$. We have

$$\begin{array}{rcl} fv(\sigma \cdot U) & = & fv((\sigma' \circ_s [x := t]) \cdot U) \\ & = & fv(\sigma' \cdot ([x := t] \cdot (\{(x,t)\} \cup U'))) \\ & = & fv(\sigma' \cdot t) \cup fv(\sigma' \cdot ([x := t]U')) \end{array}$$

Then point (i) follows from (1) and (3) by applying Lemma 1. Points (ii) and (iii) follow from (2) and (3) and the facts that $dom(\sigma) \subseteq dom(\sigma') \cup \{x\}$ and $vran(\sigma) \subseteq vran(\sigma') \cup fv(t)$.

- Case **Simp**, i.e., $x = t$. The induction hypothesis for this case states that the lemma holds for $U'$ and any $\sigma'$. Since $\textsc{unify}(U) = \textsc{unify}(U')$, we can immediately apply the induction hypothesis for $\sigma' = \sigma$ and derive that (i)-(iii) hold for $U'$ and $\sigma$. We also have that $fv(U) = fv(U') \cup \{x\}$. Points (ii) and (iii) then follow directly from the induction hypothesis. For point (i), we observe that $fv(\sigma \cdot U) = fv(\sigma(x)) \cup fv(U')$. From Lemma 1, we derive that $fv(\sigma(x)) \subseteq vran(\sigma)$. Hence, $fv(\sigma \cdot U) \subseteq fv(U)$ as required for point (i).
- Case **Occurs**, i.e., $x \in fv(t)$ and $x \neq t$. This case holds by contradiction with the assumption that $\textsc{unify}(U) = \sigma$.

This establishes points (i)-(iii) of the lemma.

We proceed with point (iv), which we also prove by computation induction. We only present the case **Unify**, all the other cases either hold trivially, by the induction hypothesis, or by contradictory assumptions. So suppose $x \notin fv(t)$. By the assumption $\textsc{unify}(U) = \sigma$, there is a $\sigma'$ such that $\textsc{unify}([x := t] \cdot U) = \sigma'$ and $\sigma = \sigma' \circ_s [x := t]$. Thus, by the induction hypothesis, we have $dom(\sigma') \cap vran(\sigma') = \emptyset$. Applying point (iii) to $\textsc{unify}([x := t] \cdot U) = \sigma'$, we get that $vran(\sigma') \subseteq fv([x := t] \cdot U)$. As $x \notin fv([x := t] \cdot U)$ (otherwise $x$ would be free in $t$), we obtain that $x \notin vran(\sigma')$.

Now, suppose that $z \in dom(\sigma)$ and $z \in vran(\sigma)$ for some $z$. We show that this leads to a contradiction. As $dom(\sigma) \subseteq dom(\sigma') \cup dom([x := t])$ and $z \in dom(\sigma)$, we know that $z \in dom(\sigma') \cup \{x\}$. Similarly, $z \in vran(\sigma') \cup fv(t)$ as $z \in vran(\sigma)$ and $vran(\sigma) \subseteq vran(\sigma') \cup vran([x := t])$. Hence, $z \in dom(\sigma')$, because $dom(\sigma') \cap vran(\sigma') = \emptyset$ and $x \notin fv(t)$ and $x \notin vran(\sigma')$. Thus, it suffices to show that $z \in vran(\sigma')$, because this contradicts the induction hypothesis $dom(\sigma') \cap vran(\sigma') = \emptyset$. By the definition of $vran(\sigma)$, there must be some $y \in dom(\sigma)$ such that $z \in fv(\sigma \cdot y)$. We distinguish two cases: whether $y = x$ or not. If $y = x$, then $\sigma \cdot y = \sigma' \cdot t$, so $z \in fv(\sigma' \cdot t)$. By Lemma 1, $z \in (fv(t) \setminus dom(\sigma')) \cup vran(\sigma')$. Therefore, $z \in vran(\sigma')$ as $z \in dom(\sigma')$. If $y \neq x$, then $\sigma \cdot y = \sigma' \cdot y$ and $y \in dom(\sigma')$, so $z \in vran(\sigma')$. □

## 2.4 Assignments

The assignments below guide you through the stepwise formalization of the results about substitution and unification from this section. We only list the main lemmas below. You will have to discover and prove additional ones.

Assignments marked with (★) can be skipped in a first go by just stating the theorems and skipping the proofs with `sorry`. Replace them later with proper proofs.

**Assignment 1** (First-order term algebras and substitution)**.** Define a datatype for a general first-order term algebra parametrized by type variables *'f* for the signature and *'v* for the type of variables as follows.

**datatype_new** $('f,\ 'v)$ "*term*" $=$ *Var* $'v$ | *Fun* $'f$ "$('f,\ 'v)$ *term list* "

The type *term* does not ensure the correct arity for function symbols (Assignment 4 below will deal with this). However, the above development trivially generalises to terms without arity restrictions on function symbols.

Note that there is a separate constructor *Var* for variables, because syntactic conventions such as $x$ always denoting to a variable cannot be expressed in Isabelle. Consequently, your formalisation has to adapt the above presentation whenever variables are involved. For example, the identity substitution is not *id*, but *Var*.

(a) Define a recursive function *fv* :: "$('f,\ 'v)$ *term* $\Rightarrow$ $'v$ *set*" that computes the set of variables in a term.

(b) Define the type synonym $('f,'v)$ *subst* $=$ $'v \Rightarrow ('f,\ 'v)$ *term* for substitutions. Then define the following two functions on substitutions: *sapply* which lifts substitutions to functions on all terms and *scomp* which composes two substitutions:

*sapply* :: "$('f,'v)$ *subst* $\Rightarrow$ $('f,'v)$ *term* $\Rightarrow$ $('f,'v)$ *term*" (*infixr* "·" 67)
*scomp* :: "$('f,'v)$ *subst* $\Rightarrow$ $('f,'v)$ *subst* $\Rightarrow$ $('f,'v)$ *subst*" (*infixl* "$\circ$s" 75)

The annotations after the type declare infix function symbols for these functions, their associativity (*l*eft or *r*ight), and their precedence. As a result, we use the same notation in the in our Isabelle theories as in the preceding text, i.e., we can write $\sigma\cdot t$ and $\sigma \circ s\ \tau$.

(c) Prove the following lemmata about substitution.

**lemma** *fv_sapply*: "$fv\ (\sigma\ \cdot\ t)\ =\ (\bigcup x \in fv\ t.\ fv\ (\sigma\ x))$"

**lemma** *sapply_cong*:
  assumes "$\bigwedge x.\ x \in fv\ t \Longrightarrow \sigma\ x = \tau\ x$"
  shows "$\sigma \cdot t\ =\ \tau \cdot t$"

**lemma** *scomp_sapply*: "$(\sigma\ \circ s\ \tau)\ x = \sigma \cdot (\tau\ x)$"
**lemma** *sapply_scomp_distrib*: "$(\sigma\ \circ s\ \tau)\ \cdot\ t\ =\ \sigma \cdot\ (\tau\ \cdot\ t)$"

**lemma** *scomp_assoc*: "$(\sigma\ \circ s\ \tau)\ \circ s\ \varrho = \sigma \circ s\ (\tau\ \circ s\ \varrho)$"
**lemma** *scomp_Var* [*simp*]: "$\sigma\ \circ s\ Var = \sigma$"
**lemma** *Var_scomp* [*simp*]: "$Var\ \circ s\ \sigma = \sigma$"

(d) Define the domain and variable range of substitutions.[1]

$sdom$ :: "('$f$, '$v$) $subst$ ⇒ '$v$ $set$"
$svran$ :: "('$f$, '$v$) $subst$ ⇒ '$v$ $set$"

Prove the following lemmata:[2]

**lemma** *sdom_ Var* [*simp*]: "$sdom\ Var = \{\}$"
**lemma** *svran_ Var* [*simp*]: "$svran\ Var = \{\}$"

**lemma** *sdom_ single_ non_ trivial* [*simp*]:
"$t \neq Var\ x \Longrightarrow sdom\ (Var(x{:=}t)) = \{x\}$"
**lemma** *svran_ single_ non_ trivial* [*simp*]:
"$t \neq Var\ x \Longrightarrow svran\ (Var(x{:=}t)) = fv\ t$"

**lemma** *fv_ sapply_ sdom_ svran*:
"$x \in fv\ (\sigma\ \cdot\ t) \Longrightarrow x \in (fv\ t\ -\ sdom\ \sigma) \cup svran\ \sigma$"

**lemma** *sdom_ scomp*: "$sdom\ (\sigma \circ s\ \tau) \subseteq sdom\ \sigma \cup sdom\ \tau$"
**lemma** *svran_ scomp*: "$svran\ (\sigma \circ s\ \tau) \subseteq svran\ \sigma \cup svran\ \tau$"

**Assignment 2** (Unification problems)**.** Define a type synonym for equations (as pairs of terms) and equation systems (as lists of equations) and lift the definitions of the functions *fv* and *sapply* (substitution on terms) to equations and equation systems.

(a) State and prove versions of lemmas *fv_ sapply* and *sapply_ scomp_ distrib* for equations and equation systems.

(b) Define the predicates *unifies* and *unifiess* to express that a given substitution is a unifier for an equation (system). Moreover, define a function *is_ mgu* asserting that a given substitution is the mgu for an equation system.

(c) Prove the following lemma and a version of it for equation systems.

**lemma** *unifies_ sapply_ eq*:
"$unifies\ \sigma\ (sapply\_eq\ \tau\ eq) \longleftrightarrow unifies\ (\sigma \circ s\ \tau)\ eq$"

**Assignment 3** (Unification algorithm and its properties)**.**

(a) Define the function UNIFY from Algorithm 1 as a recursive function of the following form:

**function** (*sequential*) *unify* :: "('$f$, '$v$) $equations$ ⇒ ('$f$, '$v$) $subst\ option$"

Use pattern matching with nested patterns and pattern completeness.

(★) Establish termination using the three measure functions $\chi_1$, $\chi_2$, and $\chi_3$ from the proof of Theorem 1.

[1] We are using *sdom* and *svran* here to distinguish these from the functions *dom* and *ran* on maps that are already defined in Isabelle.

[2] The notation $f(x := y)$ changes the function $f$ such that $x$ is mapped to $y$. Use the query panel to find theorems about the underlying function *fun_ upd*.

(b) Formalize and prove the soundness theorem (Theorem 2).

Hint: Split the proof into two parts and prove them separately by computational induction.

(i) If *unify* returns a substitution, it is a unifier.

(ii) If *unify* returns a substitution $\sigma$ and there is another unifier $\tau$, then $\tau = \rho \circ_s \sigma$ for some $\rho$.

(c) (★) Formalize and prove the completeness theorem (Theorem 3).

(d) (★) Formalize and prove the properties stated in Lemma 3.

**Assignment 4** (Arities and well-formed terms)**.**

(a) Extend your theory with arities by defining a well-formedness predicate expressing that a term respects a given arity function. Lift this function to substitutions as well as to equations and equation systems (not shown below).

```
wf_term :: "('f ⇒ nat) ⇒ ('f, 'v) term ⇒  bool"
wf_subst ::  "('f ⇒ nat) ⇒ ('f, 'v) subst ⇒ bool"
```

(b) Prove the following lemmas.

```
lemma wf_term_sapply:
  "⟦ wf_term arity t; wf_subst arity σ ⟧ ⟹ wf_term arity (σ · t) "

lemma wf_subst_scomp:
  "⟦ wf_subst arity σ; wf_subst arity τ ⟧ ⟹ wf_subst arity (σ ∘s τ) "

lemma wf_subst_unify:
  "⟦ unify eqs = Some σ; wf_eqs arity eqs ⟧ ⟹ wf_subst arity σ "
```

# 3 Symbolic Verification of Security Protocols

A typical example of a security protocol is the Needham-Schröder Public-key (NSPK) protocol [3], which is informally specified as follows:

$$\begin{array}{lll} \text{M1.} & A \to B: & \{A, NA\}_B \\ \text{M2.} & B \to A: & \{NA, NB\}_A \\ \text{M3.} & A \to B: & \{NB\}_B \end{array}$$

This protocol's goal is to achieve mutual entity authentication between the initiator $A$ and the responder $B$. Moreover, the nonces $NA$ and $NB$ should remain secret (e.g., one could use them to derive a session key). The protocol uses public-key encryption, where $\{M\}_A$ denotes the encryption of message $M$ with agent $A$'s public-key.[3] The initiator $A$ starts the protocols by sending her nonce $NA$ encrypted with the responder $B$'s public key (M1). $B$ replies with the encryption for $A$ of this nonce and his own nonce $NB$ (M2). Finally, the initiator sends nonce $NB$ encrypted for $B$ (M3). The idea is that each role's nonce can only be decrypted by the other role and getting it back should therefore authenticate the peers to each other. Moreover, the use of encryption should keep the nonces secret.

In 1996, Gawin Lowe discovered a now well-known man-in-the-middle attack on this protocol [1] where $A$ talks to the intruder, but $B$ believes to be talking to $A$ whereas in reality he is talking to the intruder. Our goal is to develop a security protocol analyzer that can discover such attacks. We work in a symbolic (Dolev-Yao) security protocol model where we assume that cryptography is perfect, i.e., decryption requires the knowledge of the correct decyption key, and that the intruder controls the network, i.e., he can observe all messages, block messages, and construct and insert messages that he can build from his knowledge.

## 3.1 Security protocol model

Our security protocol model consists of a term algebra representing protocol messages, an attacker model, and an operational semantics. We define the latter only informally as we do not need it in our development.

**Messages and attacker model** Let $\mathcal{C}$ be a set of constants. The set of protocol messages that we will use is the set $\mathcal{T}_\Sigma(\mathcal{V})$ of first-order terms generated by the following signature (using some syntactic sugar for function symbols and indicating each symbol's arity by dots):

$$\Sigma = \mathcal{C} \cup \{\mathsf{h}(\cdot), \langle\cdot,\cdot\rangle, \{\!|\cdot|\!\}_\cdot, \{\cdot\}_\cdot, [\cdot]_\cdot\}$$

Recall that constants are function symbols of arity 0. We use constants to represent agent names and nonces. We assume a special agent name $\iota \in \mathcal{C}$

[3]We in fact identify the agent's identity with his public key here.

**Axiom rule**

$$\frac{u \in T}{T \vdash u}\ \mathsf{Ax}$$

**Composition rule**

$$\frac{T \vdash t_1 \quad \cdots \quad T \vdash t_{ar(f)}}{T \vdash f(t_1, \ldots, t_{ar(f)})}\ \mathsf{Comp}\ (f \in \Sigma_c)$$

**Analysis rules**

$$\frac{T \vdash \langle t_1, t_2 \rangle}{T \vdash t_i}\ \mathsf{Proj}_i \qquad \frac{T \vdash \{\!|t|\!\}_k \quad T \vdash k}{T \vdash t}\ \mathsf{Sdec} \qquad \frac{T \vdash \{t\}_\iota}{T \vdash t}\ \mathsf{Adec}$$

Figure 1: Intruder deduction rules

representing the intruder. The term $\mathsf{h}(t)$ represents the hash of $t$ and the terms $\{\!|t|\!\}_k$, $\{t\}_k$, and $[t]_k$ respectively denote the symmetric encryption, public-key encryption, and the signature of term $t$ with key $k$.

The capabilities of the intruder to derive a message $t$ from a set $T$ of observed messages is expressed by a judgement of the form $T \vdash t$. The set of derivable judgements is inductively defined by the rules in Figure 1. The axiom rule $\mathsf{Ax}$ states that the intruder can derive any observed message. The composition rule $\mathsf{Comp}$ expresses the intruder's capability to construct new messages from derivable ones. We define $\Sigma_c = \{\mathsf{h}(\cdot), \langle \cdot, \cdot \rangle, \{\!|\cdot|\!\}_\cdot, \{\cdot\}_\cdot, [\cdot]_\iota\}$. Note that the intruder can only sign messages with his own name (for $f = [\cdot]_\iota$). According to the analysis rules, the intruder can project pairs to their components (rule $\mathsf{Pair}$), decrypt symmetric encryptions for which he knows the key (rule $\mathsf{Sdec}$), and decrypt asymmetric encryptions with the intruder's public key (rule $\mathsf{Adec}$). Note that we model signatures without message recovery, which is reflected in the absence of an analysis rule for signatures.

**Lemma 4.** *Intruder deduction satisfies the following properties:*

*(i) (Cut) If $T, t \vdash u$ and $T \vdash t$ then $T \vdash u$.*

*(ii) (Weakening) If $T \vdash t$ and $T \subseteq T'$ then $T' \vdash t$.*

*Proof.* Both properties are established by rule induction. □

**Role-based protocol specifications** We now sketch the operational semantics of security protocols. The Alice & Bob notation used for the example above is rather informal. For our purposes, it is useful to take a role-based view of security protocols, where each role is specified by a small program consisting of send and receive events.

**Example 3.** The roles of the NSPK protocol are specified as follows:

$$\begin{aligned} \mathit{NSPK}(A) &= \mathsf{send}(\{A, na\}_B) \cdot \mathsf{recv}(\{na, \mathit{NB}\}_A) \cdot \mathsf{send}(\{\mathit{NB}\}_B) \\ \mathit{NSPK}(B) &= \mathsf{recv}(\{A, \mathit{NA}\}_B) \cdot \mathsf{send}(\{\mathit{NA}, nb\}_A) \cdot \mathsf{recv}(\{nb\}_B) \end{aligned}$$

We henceforth adopt the convention to write variables in upper case and constants in lower case letters. The nonce $na$ is generated by $A$ and sent to $B$, hence it is a constant in role $A$ that is received into the variable $\mathit{NA}$ of role $B$. Similarly, nonce $nb$ is generated by $B$ and received into variable $\mathit{NB}$ by $A$. ♠

We require a minimal *well-formedness* condition for protocol roles, namely, that all variables except agent variables (such as $A$ and $B$) must first occur in a receive event.

**Operational semantics** Instead of giving operational semantics rules defining the transitions of the send and receive events, we describe the behavior of these events informally. To execute a protocol, we instantiate protocol roles into *threads* by subscripting all their events, variables, and constants with the thread identifier and by instantiating the agent variables with agent names (i.e., constants).

The state $(\mathit{IK}, \mathit{th})$ of the protocol consists of the current intruder knowledge $\mathit{IK}$ and a set of threads, the *thread pool th*. Each initial state represents the intruder's initial knowledge $\mathit{IK}_0$ consisting of ground terms and a *scenario* with an arbitrary but fixed number of threads (instantiated roles).

A state transition corresponds to the execution of the first event $ev$ of some thread $i$ in the pool and is labeled by $ev_i$. A send event $\mathsf{send}(t)$ adds the term $t$ to the intruder knowledge $\mathit{IK}$ and removes the event from the thread. The transition is labeled by $\mathsf{send}_i(t)$. The execution of a receive event $\mathsf{recv}(t)$ requires that the term $t\sigma$ for some substitution $\sigma$ with $dom(\sigma) = fv(t)$ is deducible from the current intruder knowledge $\mathit{IK}$, i.e., $\mathit{IK} \vdash \sigma \cdot t$. As a result, the receive event is removed from the thread and the substitution $\sigma$ is applied to the remaining events of the thread. The transition is labeled by $\mathsf{recv}_i(\sigma \cdot t)$. The sequence of transition labels generated by a series of execution steps is called a *trace*.

Note that the well-formedness condition for protocol roles ensures that the trace and the intruder knowledge only contain ground terms.

**Example 4** (NSPK threads)**.** Consider the initiator and responder threads $\mathit{th}_0$ and $\mathit{th}_1$ which are instantiations of the roles in Example 3 with respective agent assignments $\rho_0 = [A_0 := a, B_0 := \iota]$ and $\rho_1 = [A_1 := a, B_1 := b]$.

$$\begin{aligned} \mathit{th}_0 &= \mathsf{send}_0(\{a, na_0\}_\iota) \cdot \mathsf{recv}_0(\{na_0, \mathit{NB}_0\}_a) \cdot \mathsf{send}_0(\{\mathit{NB}_0\}_\iota) \\ \mathit{th}_1 &= \mathsf{recv}_1(\{a, \mathit{NA}_1\}_b) \cdot \mathsf{send}_1(\{\mathit{NA}_1, nb_1\}_a) \cdot \mathsf{recv}_1(\{nb_1\}_b) \end{aligned}$$

Suppose the intruder's initial knowledge is $\mathit{IK}_0 = \{a, b, \iota\}$. The send event in thread $\mathit{th}_0$ adds the message $\{a, na_0\}_\iota$ to the intruder knowledge $\mathit{IK}$. Since the intruder know $\iota$ and $b$, he can clearly derive $\mathit{IK} \vdash [\mathit{NA}_1 := na_0] \cdot \{a, \mathit{NA}_1\}_b$ and therefore the receive event in thread $\mathit{th}_1$ can be executed. As a result, the receive event is removed from thread $th_1$ and the rest is instantiated with the substitution $[NA_1 := na_0]$ resulting in the updated $th_1' = \mathsf{send}(\{na_0, nb_1\}_a) \cdot \mathsf{recv}(\{nb_1\}_b)$. The execution of these two events results in the following trace: $\mathsf{send}(\{a, na_0\}_\iota) \cdot \mathsf{recv}(\{a, na_0\}_b)$. ♠

**Security properties** The main security properties for security protocols are secrecy and authentication. Here, we only consider secrecy.

We say that a thread *honestly instantiates* a role $R$ if it instantiates role $R$ and no agent variable is assigned to the intruder $\iota$. A simple way to check whether a term $t$ is secret in role $R$ is to extend role $R$ with a final receive event $\mathsf{recv}(t)$. The term $t$ is secret in role $R$ if no thread honestly instantiating role $R$ can ever reach the receive event $\mathsf{recv}(t)$. The final receive event is executable if and only if the intruder can derive the (instantiated) term $t$ from its knowledge $IK$ (cf. semantics of receive events). Therefore, an *attack* on the secrecy of a term $t$ in role $R$ consists of a trace leading to a state $(th, IK)$ where there is an honestly instantiated thread $i$ in $th$ that has terminated its role.

**Example 5** (NSPK attack)**.** Gawin Lowe discovered the following well-known man-in-the-middle attack on the NSPK protocol [1]:

$$\begin{array}{lll}
\mathrm{M1}_0. & a \to \iota : & \{a, na\}_\iota \\
\mathrm{M1}_1. & \iota(a) \to b : & \{a, na\}_b \\
\mathrm{M2}_1. & b \to \iota(a) : & \{na, nb\}_a \\
\mathrm{M2}_0. & \iota(a) \to a : & \{na, nb\}_a \\
\mathrm{M3}_0. & a \to \iota : & \{nb\}_\iota \\
\mathrm{M3}_1. & \iota(a) \to b : & \{nb\}_b
\end{array}$$

There are two protocol threads in this attack. Thread 0 is run by agent $a$ in the initiator role $A$ with the intruder $\iota$. Thread 1 is an instance of responder role $B$ and is run by $b$ with $a$. The secrecy and authentication properties are only required to hold for threads that communicate with honest peers. Hence, $a$ cannot expect any security guarantees (technically, her guarantees hold trivially). Agent $b$'s guarantees clearly fail. He believes to be talking to the honest agent $a$ whereas in reality he is talking to the intruder who impersonates $a$, indicated by the notation $\iota(a)$. Moreover, the intruder can extract both nonces, $na$ and $nb$, from the messages $\mathrm{M1}_1$ and $\mathrm{M3}_1$. In particular, $a$ acts as a decryption oracle for the intruder when she extracts $nb$ from message $\mathrm{M2}_1$.

The scenario of Lowe's attack corresponds to the interleaved execution of the threads $th_0$ and $th_1$ from Example 4. We express the secrecy of the nonces $NA$ and $nb$ for role $B$ by adding the event $\mathsf{recv}(\langle NA, nb\rangle)$ to role $B$, resulting in the extension of thread $th_1$ with the event $\mathsf{recv}(\langle NA_1, nb_1\rangle)$. Assuming $IK_0 = \{a, b, \iota\}$, the following trace represents Lowe's attack on the NSPK protocol.

$$\begin{array}{l}
\mathsf{send}_0(\{a, na_0\}_\iota) \cdot \mathsf{recv}_1(\{a, na_0\}_b) \cdot \mathsf{send}_1(\{na_0, nb_1\}_a) \cdot \mathsf{recv}_0(\{na_0, nb_1\}_a) \cdot \\
\mathsf{send}_0(\{nb_1\}_\iota) \cdot \mathsf{recv}_1(\{nb_1\}_b) \cdot \mathsf{recv}_1(\langle na_0, nb_1\rangle)
\end{array}$$

To check that this is indeed a trace of the augmented NSPK protocol, note that the intruder knowledge at the end of the trace is $IK_0$ augmented with the messages from the send events, i.e.,

$$IK = IK_0 \cup \{ \{a, na_1\}_\iota, \{na_0, nb_1\}_a, \{nb_1\}_i \}$$

Clearly, the intruder can derive the pair of nonces $\langle na_0, nb_1 \rangle$ from $IK$. ♠

## 3.2 Constraint-based protocol analysis

The problem of verifying secrecy properties of security protocols is undecidable without a bound on the number of protocol threads or on the size of messages the intruder may generate. For a bounded number of threads the problem becomes decidable [4]. However, even in this case the transition systems generated by the operational semantics generally have infinitely many states. The problem stems from the fact that in each receive event the intruder can potentially send infinitely many messages (their size is unbounded). This leads to transition systems that are generally infinitely branching although only of finite depth. As a concrete example, in the NSPK protocol, an agent $b$ in role $B$ would accept the message $\{a, \mathsf{h}^k(\iota)\}_b$ for any $k \in \mathbb{N}$ as the first message and interpret $\mathsf{h}^k(\iota)$ as $a$'s nonce.

### 3.2.1 Constraint systems

To address this problem and obtain a practical decision procedure, Millen and Shmatikov [2] have proposed an analysis method based on constraint solving, which subsequently has been refined and extended many ways. Their method reduces the branching factor to at most one successor for each thread. The basic idea is to consider *symbolic* protocol executions where the threads' variables remain uninstantiated. For each receive event $\mathsf{recv}(t)$ on such a trace, we record a derivability constraint $IK \vdash t$ where $IK$ consists of the (symbolic) messages appearing in the preceding send events.

The desired relation between symbolic and ground execution traces is as follows. Given a symbolic trace $tr$, the resulting constraint system $cs$ has a solution $\sigma$ (ground substitution) such that the instance $\sigma \cdot IK \vdash \sigma \cdot t$ of each constraint $IK \vdash t$ in $cs$ becomes intruder-deducible (Figure 1) if and only if the trace $\sigma \cdot tr$ corresponds to a ground execution of the protocol.

**Example 6.** The symbolic trace corresponding to Lowe's attack on the NSPK protocol and its corresponding constraint system are as follows.

$$\begin{array}{ll}
 & IK_0 \vdash \langle A_0, B_0, A_1, B_1 \rangle \\
\mathsf{send}_0(\{A_0, na_0\}_{B_0})\cdot & IK_1 = IK_0, \{A_0, na_0\}_{B_0} \\
\mathsf{recv}_1(\{A_1, NA_1\}_{B_1})\cdot & IK_1 \vdash \{A_1, NA_1\}_{B_1} \\
\mathsf{send}_1(\{NA_1, nb_1\}_{A_1})\cdot & IK_2 = IK_1, \{NA_1, nb_1\}_{A_1} \\
\mathsf{recv}_0(\{na_0, NB_0\}_{A_0})\cdot & IK_2 \vdash \{na_0, NB_0, \}_{A_0} \\
\mathsf{send}_0(\{NB_0\}_{B_0})\cdot & IK_3 = IK_2, \{NB_0\}_{B_0} \\
\mathsf{recv}_1(\{nb_1\}_{B_1})\cdot & IK_3 \vdash \{nb_1\}_{B_1} \\
\mathsf{recv}_1(\langle NA_1, nb_1 \rangle) & IK_3 \vdash \langle NA_1, nb_1 \rangle
\end{array}$$

The symbolic trace interleaves a thread 0 instantiating role $A$ and a thread 1 instantiating role $B$. Note that we have not instantiated the agent variables here. In order to check the secrecy of $NA_1$ and $nb_1$ we will have to instantiate $A_1$ and $B_1$ with honest agents (cf. definition of secrecy).

The constraint system is composed of five constraints. The first constraint encodes the fact that the agent names instantiating the agent variables $A_0$, $B_0$, $A_1$, and $B_1$ are known to the intruder. The remaining constraints stem from the trace events. There is one constraint for each receive event. The left-hand side of each constraint consists of the set of terms composed of the initial intruder knowledge augmented with the terms from the send events preceding the given receive event. The term from the receive event appears on the right-hand side. The final constraint checks the violation of the secrecy of nonces $NA_1$ and $nb_1$. Lowe's attack corresponds to the substitution

$$\sigma = [A_0 := a, B_0 := \iota, A_1 := a, B_1 := b, NB_0 := nb_1, NA_1 := na_0]$$

which instantiates the symbolic trace above to the attack trace of Example 5 and solves the constraint system above, i.e., all constraints instantiated with $\sigma$ can be derived using the intruder deduction rules from Figure 1. ♠

We now formally define constraint systems and their semantics. We adopt a slightly refined constraint format, which will help us with the proof that the constraint solving process terminates.

**Definition 1.** An intruder deduction *constraint* $M \,|\, A \rhd t$ consists of two finite sets of terms $M$ and $A$ and a term $t$. A *constraint system* $cs$ is a finite set of constraints.

*Notation.* For the sake of readability, we introduce a lightweight notation for certain set operations. We will often write a comma for union and drop the curly brackets around singleton sets. For example, $M, t \,|\, A, B \rhd u$ denotes the constraint $c = M \cup \{t\} \,|\, A \cup B \rhd u$ and similarly for constraints and constraint systems.

A solution of a constraint system $cs$ is a substitutions that makes all constraints in $cs$ intruder-derivable. Note that this ignores the separation of $M$ and $A$, which only plays a role in the constraint solving process.

**Definition 2.** The *solution set* of a constraint system $cs$ is defined by

$$sol(cs) = \{\sigma \mid \forall (M \,|\, A \rhd t) \in cs.\ \sigma \cdot (M, A) \vdash \sigma \cdot t\}.$$

A constraint system is *satisfiable* if $sol(cs) \neq \emptyset$, i.e., there exists a solution, and *unsatisfiable* otherwise.

The following lemmas record properties of the solution sets of constraint systems regarding composition and substitution.

**Lemma 5.** *$sol(cs_1, cs_2) = sol(cs_1) \cap sol(cs_2)$.*

**Lemma 6.** *If $\tau \in sol(\sigma \cdot cs)$ then $\tau \circ_s \sigma \in sol(cs)$.*

In the remainder of this section, we present constraint reduction rules to solve constraint systems and their desired properties.

**Unification rule**

$$\mathsf{Unif}^{\ell} \quad M \,|\, A \rhd t \quad \rightsquigarrow^{1}_{\sigma} \quad \emptyset \quad \text{if } t \notin \mathcal{V},\ u \in M \cup A, \text{ and } \sigma = \textsc{unify}(t,u)$$

**Composition rule ($f \in \Sigma_c$)**

$$\mathsf{Comp}^{\ell} \quad M \,|\, A \rhd f(t_1, \ldots, t_{ar(f)}) \quad \rightsquigarrow^{1}_{id} \quad \{M \,|\, A \rhd t_1,\ \ldots,\ M \,|\, A \rhd t_{ar(f)}\}$$

**Analysis rules**

$$\begin{array}{llll}
\mathsf{Proj}^{\ell} & M, \langle u, v\rangle \,|\, A \rhd t & \rightsquigarrow^{1}_{id} & \{M, u, v \,|\, \langle u, v\rangle, A \rhd t\} \\
\mathsf{Sdec}^{\ell} & M, \{\!|u|\!\}_k \,|\, A \rhd t & \rightsquigarrow^{1}_{id} & \{M, u \,|\, \{\!|u|\!\}_k, A \rhd t,\ M \,|\, \{\!|u|\!\}_k, A \rhd k\} \\
\mathsf{Adec}^{\ell} & M, \{u\}_{\iota} \,|\, A \rhd t & \rightsquigarrow^{1}_{id} & \{M, u \,|\, \{u\}_{\iota}, A \rhd t\} \\
\mathsf{Ksub}^{\ell} & M, \{u\}_x \,|\, A \rhd t & \rightsquigarrow^{1}_{[x:=\iota]} & \{[x := \iota] \cdot (M, \{u\}_x \,|\, A \rhd t)\}
\end{array}$$

Figure 2: Constraint reduction rules

### 3.2.2 Constraint solving

Figure 2 defines constraint reduction relations of the form $\rightsquigarrow^{1}_{\sigma}$ (indexed by substitutions) between a constraint and a constraint system. The intention is that the constraint on the left is replaced by the (possibly empty) constraint system on the right and the substitution $\sigma$ is applied to the whole constraint system. This is expressed in the following rule, which extends each relation $\rightsquigarrow^{1}_{\sigma}$ to the relation $\rightsquigarrow_{\sigma}$ on constraint systems:

$$\frac{c \rightsquigarrow^{1}_{\sigma} cs}{c, cs' \rightsquigarrow_{\sigma} cs,\ \sigma \cdot cs'}\ \textbf{Context}$$

We now briefly describe each constraint reduction rule and its relation to the intruder deduction rules of Figure 1. The unification rule $\mathsf{Unif}^{\ell}$ unifies a term $u$ on the left-hand side of a constraint with the term $t$ on the right hand side, which must not be a variable (we motivate this condition below). The corresponding mgu $\sigma$ labels the reduction step (and is applied to the other constraints of the system, see rule **Context**). This rule is a generalization of the axiom rule Ax. The composition rule $\mathsf{Comp}^{\ell}$ removes the top-level symbol $f \in \Sigma_c$ of the right-hand side term of a constraint and replaces the constraint by one constraint for each argument term $t_i$ of $f$. This rule directly reflects the intruder deduction rule Comp.

The analysis rules $\mathsf{Proj}^{\ell}$, $\mathsf{Sdec}^{\ell}$, and $\mathsf{Adec}^{\ell}$ decompose a pair or a ciphertext $t$ in the set $M$ on the left-hand side of a constraint $M \mid A \rhd t$. As a result, the analyzed term $t$ is moved to the set $A$ (to avoid repeated analysis of the same term) and the term's content (pair components or plaintext) is added to $M$. The decryption rule $\mathsf{Sdec}^{\ell}$ generates an additional constraint requiring the derivation of the appropriate decryption key. The analysis rule $\mathsf{Adec}^{\ell}$ decrypts an asymmetric encryption with the intruder's public key $\iota$. The rule $\mathsf{Ksub}^\ell$ prepares the application of rule $\mathsf{Adec}^\ell$ in the case where the public-key is a variable $x$. This rule instantiates $x$ with $\iota$ in the constraint (and the surrounding constraint system, see rule **Context**). While the analysis rules $\mathsf{Proj}^\ell$, $\mathsf{Sdec}^\ell$, and $\mathsf{Adec}^\ell$ are justified by the corresponding intruder deduction rules $\mathsf{Proj}_i$, $\mathsf{Sdec}$, $\mathsf{Adec}$, they do not directly correspond to them, since they operate on the left-hand side of constraints (see Section 3.2.3).

**Finding solutions** The relations $\rightsquigarrow_\sigma$ describe a single reduction step between constraint systems. We define the reflexive and transitive closure $\rightsquigarrow^*_\cdot$ of these relations as the smallest relation satisfying the following properties:

(i) $cs \rightsquigarrow^*_{id} cs$, and

(ii) $cs \rightsquigarrow^*_{\tau \circ_s \sigma} cs'$ whenever $cs \rightsquigarrow_\sigma cs''$ and $cs'' \rightsquigarrow^*_\tau cs'$ for some $cs''$.

We start the constraint solving process in an initial constraint system $cs$ that we have obtained from a symbolic trace of the protocol under study. We repeatedly apply the reduction rules to obtain reduction sequences $cs \rightsquigarrow^*_\sigma cs'$ for some substitution $\sigma$ and constraint system $cs'$. We can stop the reduction at so-called *simple* constraint systems $cs'$ where all constraints have variables on their right-hand sides, i.e., they are of the form

$$M \,|\, A \rhd X.$$

The intuition is that the intruder can replace these variables with any term that he can construct. Hence, under the realistic assumption that the intruder's initial knowledge is non-empty, *every simple constraint system has a solution.* The fact that we can stop the reduction at simple constraint systems exactly avoids the infinite branching that would arise by instantiating the variables as in the ground operational semantics.

We now define the set of substitutions obtained by reduction of a constraint system $cs$ to simple ones.

**Definition 3.** We define the set of *reducts* of a constraint system $cs$ by

$$red(cs) = \{\tau \circ_s \sigma \mid \exists cs'.\ cs \rightsquigarrow^*_\sigma cs' \wedge cs' \text{ is simple} \wedge \tau \in sol(cs')\}.$$

**Overview of properties** We now turn to the three main properties of constraint reduction: soundness, completeness, and termination. Soundness means that all substitutions resulting from constraint reduction are indeed solutions, i.e., $red(cs) \subseteq sol(cs)$. Completeness means that the reduction does not miss any solution, i.e., $sol(cs) \subseteq red(cs)$. As a consequence, we can safely discard any irreducible non-simple constraint system $cs'$ obtained from $cs$. Termination means that there is no infinite reduction sequence, i.e., that the relation $\rightsquigarrow$ is well-founded. In the following, we discuss soundness and termination in detail, while we only state the completeness result without presenting its proof.

### 3.2.3 Soundness

The proof of the soundness result is based on the following lemma expressing one-step reduction soundness.

**Lemma 7.** *If $c \rightsquigarrow^1_\sigma cs$ and $\tau \in sol(cs)$ then $\tau \circ_s \sigma \in sol(\{c\})$.*

*Proof.* By case analysis on the reduction rule $R$ of Figure 2.

- $R = \mathsf{Unif}^\ell$. We have $c = M \mid A \rhd t$, $cs = \emptyset$, $u \in M \cup A$, and $\sigma = \mathrm{unify}(t, u)$. Then $\sigma \cdot t = \sigma \cdot u$ and therefore we have
$$\tau \cdot \sigma \cdot (M, A) \vdash \tau \cdot \sigma \cdot t.$$
for all $\tau$ by the axiom rule $\mathsf{Ax}$. This shows that $\tau \circ_s \sigma \in sol(\{c\})$.

- $R = \mathsf{Sdec}^\ell$. In this case, we have $c = M, \{\!|u|\!\}_k \mid A \rhd t$, $\sigma = id$, and $cs = \{M, u \mid \{\!|u|\!\}_k, A \rhd t,\ M \mid \{\!|u|\!\}_k, A \rhd k\}$. By the second hypothesis, $\tau \in sol(cs)$, we have
$$\tau \cdot (M, A, u, \{\!|u|\!\}_k) \vdash \tau \cdot t \tag{4}$$
$$\tau \cdot (M, A, \{\!|u|\!\}_k) \vdash \tau \cdot k \tag{5}$$
Now, we derive
$$\dfrac{\dfrac{}{\tau \cdot (M, A, \{\!|u|\!\}_k) \vdash \tau \cdot \{\!|u|\!\}_k}\,\mathsf{Ax} \qquad \dfrac{}{\tau \cdot (M, A, \{\!|u|\!\}_k) \vdash \tau \cdot k}\,(\text{by } (5))}{\tau \cdot (M, A, \{\!|u|\!\}_k) \vdash \tau \cdot u}\,\mathsf{Sdec}$$
From the conclusion of this derivation and (4) we derive $\tau \cdot (M, A, \{\!|u|\!\}_k) \vdash \tau \cdot t$ using Lemma 4(i). Hence, $\tau \circ_s \sigma \in sol(\{c\})$ as required.

We leave the remaining cases to the reader. □

The following two lemmas lift the previous one to the relations $\rightsquigarrow_\sigma$ and $\rightsquigarrow^*_\sigma$.

**Lemma 8.** *If $cs \rightsquigarrow_\sigma cs'$ and $\tau \in sol(cs')$ then $\tau \circ_s \sigma \in sol(cs)$.*

*Proof.* From $cs \rightsquigarrow_\sigma cs'$ we know from rule **Context** that $cs = c, cs_2$, $c \rightsquigarrow^1_\sigma cs_1$, and $cs' = cs_1, \sigma \cdot cs_2$. From the assumption $\tau \in sol(cs_1, \sigma \cdot cs_2)$, we derive $\tau \in sol(cs_1)$ and $\tau \in sol(\sigma \cdot cs_2)$ by Lemma 5. Therefore, we have $\tau \circ_s \sigma \in sol(\{c\})$ by Lemma 7 and $\tau \circ_s \sigma \in sol(cs_2)$ by Lemma 6. A final application of Lemma 5 yields the desired result. □

**Lemma 9.** *If $cs \rightsquigarrow^*_\sigma cs'$, $cs'$ is simple, and $\tau \in sol(cs')$ then $\tau \circ_s \sigma \in sol(cs)$.*

*Proof.* By rule induction on the definition of $\rightsquigarrow^*$. The base case is trivial since $cs = cs'$ and $\sigma = id$. In the inductive case, suppose $cs \rightsquigarrow_\rho cs'$, $cs' \rightsquigarrow^*_{\rho'} cs''$, $cs''$ is simple, and $\tau \in sol(cs'')$. From the induction hypothesis, we get $\tau \circ_s \rho' \in sol(cs')$. By Lemma 8, it follows that $\tau \circ_s (\rho' \circ_s \rho) \in sol(cs')$ as required. □

Now soundness becomes a corollary of the previous lemma.

**Theorem 4.** (Soundness) *Constraint solving is sound, i.e., $red(cs) \subseteq sol(cs)$.*

*Proof.* By the definition of $red(cs)$ and Lemma 9. □

### 3.2.4 Termination

We establish the termination of the constraint reduction process by showing that the constraint reduction relation $\rightsquigarrow = \bigcup_\sigma \rightsquigarrow_\sigma$ is well-founded. It is sufficient to show that whenever $cs \rightsquigarrow_\sigma cs'$ then $cs' \sqsubset cs$ for a well-founded relation $\sqsubset$ on constraint systems. We define the relation $\sqsubset$ as the lexicographic composition of two measure functions $\eta_1$ and $\eta_2$ on constraint systems. The measure function $\eta_2$ is constructed from two auxiliary functions on terms, $\theta$ and $\chi$:

$$\begin{array}{rcll} \theta(f(t_1,\dots,t_n)) & = & \theta(t_1)+\cdots+\theta(t_n)+1 & \text{for } f\in\Sigma_c \\ \theta(t) & = & 1 & \text{for all other terms } t \end{array}$$

$$\begin{array}{rcll} \chi(\mathsf{h}(t)) & = & \chi(t)+1 & \\ \chi(\langle t,u\rangle) & = & \chi(t)\cdot\chi(u)+1 & \\ \chi(\{\![t]\!\}_k) & = & \chi(t)+\theta(k)+1 & \\ \chi(\{t\}_k) & = & \chi(t)+1 & \\ \chi([t]_k) & = & \chi(t)+1 & \\ \chi(t) & = & 1 & \text{for all other terms } t \end{array}$$

We lift $\chi$ to finite sets of terms by defining $\chi(M) = \Pi_{u\in M}\,\chi(u)$ and combine $\chi$ and $\theta$ into a "weight" function on constraints:

$$w(M\,|\,A \rhd t) = \chi(M)\cdot\theta(t)$$

Note that the already analyzed terms in $A$ do not contribute to the weight of a constraint. This point is essential for the termination proof. We define the two measure functions $\eta_1$ and $\eta_2$ on constraint systems $cs$ as follows.

$$\begin{array}{rcl} \eta_1(cs) & = & |fv(cs)| \\ \eta_2(cs) & = & \Sigma_{c\in cs} w(c) \end{array}$$

Reductions with non-trivial substitutions (using rules $\mathsf{Unif}^\ell$ or $\mathsf{Ksub}^\ell$) decrease $\eta_1$, while those with trivial substitutions decrease $\eta_2$ without increasing $\eta_1$. This is summarized in the following table.

| | $\eta_1(cs)$ | $\eta_2(cs)$ |
|---|---|---|
| $\sigma \neq id$ | $<$ | |
| $\sigma = id$ | $\leq$ | $<$ |

The following three lemmas suffice to prove this.

**Lemma 10.** *If $c \rightsquigarrow^1_\sigma cs$ then $fv(cs, \sigma\cdot cs') \subseteq fv(c, cs')$.*

*Proof.* By case analysis on the rule $R$ used to justify $c \rightsquigarrow^1_\sigma cs$. The conclusion holds trivially for all rules except $\mathsf{Unif}^\ell$ and $\mathsf{Ksub}^\ell$, since these clearly do not introduce fresh variables. We consider here the case $R = \mathsf{Unif}^\ell$ where we have $cs = \emptyset$. From Lemma 1, we derive $fv(\sigma\cdot cs') \subseteq (fv(cs')\backslash dom(\sigma)) \cup vran(\sigma)$. From Lemma 3(iii), we know that $vran(\sigma) \subseteq fv(c)$. Hence, $fv(c, cs') \subseteq fv(cs, \sigma\cdot cs)$ as required. □

**Lemma 11.** *If $c \rightsquigarrow^1_\sigma cs$ and $\sigma \neq id$ then $fv(cs, \sigma \cdot cs') \neq fv(c, cs')$.*

*Proof.* By case analysis on the rule $R$ used to justify $c \rightsquigarrow^1_\sigma cs$. Only the rules $\mathsf{Unif}^\ell$ and $\mathsf{Ksub}^\ell$ can produce a non-trivial substitution $\sigma$.

We consider here the case $R = \mathsf{Unif}^\ell$ where we have $cs = \emptyset$. We pick an arbitrary $x \in dom(\sigma)$. By Lemma 3(ii), we have $dom(\sigma) \subseteq fv(c)$, hence $x \in fv(c, cs')$. From Lemma 1, we derive $fv(\sigma \cdot cs') \subseteq (fv(cs') \setminus dom(\sigma)) \cup vran(\sigma)$ and from Lemma 3(iv), we have $dom(\sigma) \cap vran(\sigma) = \emptyset$. Hence, $x \notin fv(cs, \sigma \cdot cs')$, which establishes the required inequality. □

**Lemma 12.** *If $c \rightsquigarrow^1_{id} cs$ then $\eta_2(cs) < w(c)$.*

*Proof.* By case analysis on the rule $R$ used to justify $c \rightsquigarrow^1_\sigma cs$. The verification of the conclusion requires some calculation. Note that $w(c) = \eta_2(\{c\})$ and $w(c) > 0$ since it is a product of positive factors.

- $R = \mathsf{Unif}^\ell$. This is easy, since $\eta_2(\emptyset) = 0 < w(c)$.
- $R = \mathsf{Proj}^\ell$. Here, $\eta_2(cs) < w(c)$ follows from $\chi(t) \cdot \chi(u) < \chi(\langle t, u \rangle)$.
- $R = \mathsf{Sdec}^\ell$. The following calculation establishes the desired property:

$$\begin{array}{rcl}
\eta_2(cs) & = & w(M, u \,|\, \{\!|u|\!\}_k, A \rhd t) + w(M \,|\, \{\!|u|\!\}_k, A \rhd k) \\
& = & \chi(M) \cdot \chi(u) \cdot \theta(t) + \chi(M) \cdot \theta(k) \\
& < & \chi(M) \cdot \chi(u) \cdot \theta(t) + \chi(M) \cdot (\theta(k) + 1) \cdot \theta(t) \\
& = & \chi(M) \cdot (\chi(u) + \theta(k) + 1) \cdot \theta(t) \\
& = & \chi(M) \cdot \chi(\{\!|u|\!\}_k) \cdot \theta(t) \\
& = & w(M, \{\!|u|\!\}_k \,|\, A \rhd t) \\
& = & w(c)
\end{array}$$

We leave the remaining cases as an exercise. □

**Theorem 5.** (Termination) *The constraint reduction relation $\rightsquigarrow$ is well-founded.*

*Proof.* Follows easily from Lemmas 10, 11, and 12. □

### 3.2.5 Completeness

Under a reasonable well-formedness condition on the constraint system, one can show the completeness of the constraint reduction. Completeness means that every solution can be found by a suitable constraint reduction sequence.

**Theorem 6.** (Completeness) *Constraint solving is complete, i.e., $sol(cs) \subseteq red(cs)$ for well-formed constraints $cs$.*

However, this theorem is out of the scope of our formalization work and we therefore omit its proof. We refer the interested reader to [2] for more details.

## 3.3 Assignments

**Assignment 5** (Transfer of unification theory)**.** We first transfer some definitions and results from our development in Section 2. Create a new **theory** *Term* for this purpose that imports the theory from Section 2.4. The idea is that all relevant results are transferred such that no linking definitions need to be unfolded outside this theory later in the development.

(a) Formalize the message term algebra as a datatype *msg* with **datatype_new**.

Hints: Define separate constructors for variables and constants that take the name of the variable or constant as argument. We suggest to use strings (type *string*) as variables and constants and not to use subscripting for keys in cryptographic operations.

(b) Define a non-recursive data type *symbol* representing the the signature (i.e., the function symbols) used by message terms. Define a function *arity* :: *symbol* ⇒ *nat* denoting the arity of each message constructor.

(c) Embed messages into the type of general terms with function symbols taken from *symbol* such that embedded messages respect the arity constraints given by *arity*. To that end, define a pair of functions

*embed* :: "*msg* ⇒(*symbol*, ...) *term*"
*msg_of_term* :: "(*symbol*, ...) *term* ⇒ *msg*"

(where ... denotes the type of variables you chose in 5a), and prove that the functions satisfy the three properties of an embedding:

**lemma** *wf_term_embed* [*simp*]: "*wf_term arity* (*embed msg*)"
**lemma** *msg_of_term_embed* [*simp*]: "*msg_of_term* (*embed msg*) = *msg*"
**lemma** *embed_msg_of_term* [*simp*]:
  "*wf_term arity t* $\Longrightarrow$ *embed* (*msg_of_term t*) = *t*"

Derive the following properties about the embedding:

**lemma** *wf_subst_embed* [*simp*]: "*wf_subst arity* (*embed* ∘ σ)"

**lemma** *msg_of_term_inject*:
  "⟦ *wf_term arity t1*; *wf_term arity t2* ⟧
  $\Longrightarrow$ *msg_of_term t1* = *msg_of_term t2* ⟷ *t1* = *t2*"

(d) Transfer the functions *fv*, *sapply*, *unifies* and *unify* to message terms by defining them via the embedding. Remember to transfer the properties of these functions, too, if you need them.

(e) Transfer the theorem that unification returns a unifier from Assignment 3bi to message terms.

(f) Transfer your formalization of Lemma 3 from Section 2 to message terms.

**Assignment 6** (Intruder deduction)**.** Create a new **theory** *Deduction* for the formalization of Section 3.

(a) Formalize the intruder deduction relation defined in Figure 1 as an inductive predicate.

**inductive** *deduce* :: "*msg set* ⇒ *msg* ⇒ *bool*"  (*infix* "⊢ " 72)

Here are some examples with which you can test your definition.

- $\{|m|\}_x, x \vdash m$
- $\langle m, n\rangle \vdash \mathsf{h}(\langle n, m\rangle)$
- $\{|m|\}_k, \{k\}_\iota \vdash \langle m, [m]_\iota\rangle$

(b) Prove the following lemmas by rule induction:

```
lemma deduce_weaken:
  assumes "G ⊢ t" and "G ⊆ H"
  shows "H ⊢ t"

lemma deduce_cut:
  assumes "insert t H ⊢ u" and "H ⊢ t"
  shows "H ⊢ u"
```

**Assignment 7** (Constraint systems and constraint reduction)**.** We formalize constraints and constraint systems (Definition 1) using lists for all finite sets.

```
datatype_new constraint =
  Constraint "msg list" "msg list" "msg"  ("((2_/|_)/ ▷_)" [67,67,67]66)
```

Note that a list imposes an order on its elements, unlike a finite set. The presentation in the previous section assumes that there is no such order. Consequently, your formalisation must account for this change of representation.

Hint: First, you should set for yourself a convention on how to deal with the order of elements. Then, stick to this convention in all of your development. For example, if you decide to always add inserted elements to the front of the list, then all your definitions should do so and your statements should also exploit this fact. If your statement requires a different order than your definitions, its proof will be long-winded and technical.

(a) Lift the notions of free variables and substitutions as well as the main lemmas about them to constraints and constraint systems.

(b) Formalize the set *sol* of solutions (Definition 2) and prove Lemmas 5 and 6 about it.

(c) Formalize the constraint reduction relations $\rightsquigarrow^1_\sigma$ from Figure 2, the relations $\rightsquigarrow_\sigma$ defined by the rule **Context**, and the transitive closures $\rightsquigarrow^*_\sigma$. Here are the types and suggested syntactic sugar for these relations.

```
inductive rer1 :: "constraint ⇒ subst ⇒ constraint_system ⇒ bool"
  ("(_)/ ⇝\<^sub>1[_]/ (_)" [64,64,64]63)
inductive rer :: "constraint_system ⇒subst ⇒ constraint_system ⇒ bool"
  ("_/ ⇝[_]/ _" [73,73,73]72)
inductive rer_star :: "constraint_system ⇒subst ⇒  constraint_system ⇒ bool"
  ("_/ ⇝*[_]/ _" [73,73,73]72)
```

Hint: Use a separate composition rule for each $f \in \Sigma_c$.

(d) Define a predicate that characterizes the simple constraints and its extension to constraint systems. Then define the set *red* of reducts (Definition 3).

**Assignment 8** (Properties of constraint reduction)**.**

(a) Prove the soundness of the reduction relation, i.e., that $red(cs) \subseteq sol(cs)$ (Theorem 4).

(b) (★) Prove the well-foundedness of the reduction relation $\rightsquigarrow$ (Theorem 5).

**Assignment 9** (Implementation (★))**.** Implement the constraint solving process as a functional program. To that end, create a new **theory** *Execute* for the implementation.

(a) Implement functions which, when given a constraint (system), compute the list of all possible successor constraint systems and the associated substitution under $\rightsquigarrow_1$ and $\rightsquigarrow$.

(b) Implement the constraint reduction process as a function *search* such that *search cs* returns *Some* (*cs*', $\sigma$) consisting of a simple constraint system $cs'$ and a substitution $\sigma$ such that *cs* $\rightsquigarrow*[\sigma]$ *cs*' if it exists and fails with *None* otherwise. Use the functions from 9a.

```
function search :: "constraint_system ⇒(constraint_system × subst) option"
```

For the termination proof use the result from the previous assignment that the one-step reduction relation $\rightsquigarrow$ is well-founded.

Hint: *search* can stop exploring states as soon as it has found any simple constraint system. It does not have to explore the whole search tree.

(c) Test your search procedure on the following examples:

**Key transport protocol**

$$\begin{aligned}
a, b, \iota &\mid \rhd \langle A_0, B_0 \rangle \\
\{\langle k_0, [k_0]_{A_0} \rangle\}_{B_0}, a, b, \iota &\mid \rhd \{\langle K_1, [K_1]_a \rangle\}_b \\
\{\!|m_1|\!\}_{K_1}, \{\langle k_0, [k_0]_{A_0} \rangle\}_{B_0}, a, b, \iota &\mid \rhd \{\!|Z_0|\!\}_{k_0} \\
\{\!|m_1|\!\}_{K_1}, \{\langle k_0, [k_0]_{A_0} \rangle\}_{B_0}, a, b, \iota &\mid \rhd \langle K_1, m_1 \rangle
\end{aligned}$$

**Needham-Schröder Public-Key protocol**

$$\begin{aligned}
a, b, \iota &\mid \rhd\langle A_0, B_0\rangle \\
\{\langle na_0, A_0\rangle\}_{B_0}, a, b, \iota &\mid \rhd\{\langle NA_1, a\rangle\}_b \\
\{\langle NA_1, nb_1\rangle\}_a, \{\langle na_0, A_0\rangle\}_{B_0}, a, b, \iota &\mid \rhd\{\langle na_0, NB_0\rangle\}_{A_0} \\
\{NB_0\}_{B_0}, \{\langle NA_1, nb_1\rangle\}_a, \{\langle na_0, A_0\rangle\}_{B_0}, a, b, \iota &\mid \rhd\{nb_1\}_b \\
\{NB_0\}_{B_0}, \{\langle NA_1, nb_1\rangle\}_a, \{\langle na_0, A_0\rangle\}_{B_0}, a, b, \iota &\mid \rhd\langle NA_1, nb_1\rangle
\end{aligned}$$

Hint: Results are easier to read if you import the theories *~~/src/HOL/Library/Code_ Target_ Nat* and *~~/src/HOL/Library/Code_ Char*.

**Assignment 10** (Properties of implementation (★))**.** Finally, we prove properties about our implementation.

(a) Prove that your search procedure is sound with respect to the constraint reduction relation.

(b) Prove that your search procedure is complete with respect to the constraint reduction relation.

## 4 General advice

Here are some points of general advice that should help you with the development:

- Be careful about abstraction. Abstraction is very helpful in keeping proof goals manageable. In particular, the use of **fun** for defining non-recursive functions without pattern matching provides no abstraction, in contrast to definitions.

- Define introduction, elimination, and destruction rules for the important definitions. This allows you to reason about the defined objects without the need to unfold their definition each time.

- Turn lists into sets where you can. For example, avoid the use of *list_ all*.

- Use library functions instead of introducing trivial recursive definitions. For example, use *list_ sum* instead of redefining it.

- You should guide the proofs in the direction you want rather than letting the tool drive you in some direction. Think about what are appropriate reasoning steps in the proof and use intermediate goals (with **have**) accordingly. Do not let the tool drive you into taking meaningless micro-steps (which also makes the resulting proofs hard to read).

- When a proof starts becoming too complicated, think about how to raise the level of abstraction, e.g., by proving appropriate lemmas.

- Try to find abstract and general lemmas. These are often easier to prove and more reusable than very specialized lemmas that only fit your precise application. Think twice about the suitable formulation of a lemma.
- Use the final proof method (with **qed**) to cover proofs of easy cases.

## 5 Project deliverables

The following deliverables are due:

**Review of main definitions** The definitions of the unification algorithm and of the constraint reduction system must be shown to the lecturers during one of the Tuesday sessions (or be sent to them by e-mail). This review shall ensure that the specification are correct and not overly complicated. We recommend that each team do this **as soon as the definitions are finished.**

**Formalisation** The complete Isabelle2015 development of each group must be handed in **on Dec 18, 2015** by e-mail to the lecturers. All sources must pass Isabelle/HOL's proof checking. The use of **sorry** is fine, but results in a deduction of points.

**Presentation** A 10 minute presentation **on Dec 15, 2015**.

The project and the exam will respectively contribute 40% and 60% to the final grade for this course. Details about the final presentation and our grading scheme can be found on the course website.

## References

[1] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software — Concepts and Tools*, 17:93–102, 1996.

[2] J. K. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *ACM Conference on Computer and Communications Security*, pages 166–175, 2001.

[3] R. Needham and M. D. Schroeder. Using encryption for authentication in large data networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[4] M. Rusinowitch and M. Turuani. Protocol insecurity with finite number of sessions is NP-complete. In *CSFW*, pages 174–. IEEE Computer Society, 2001.