Theorem Proving in Isabelle

Lutz Schröder

January 17, 2005

Unification

What is Unification?

What is Unification?

- In a nutshell: the process of finding a substitution that makes two given terms equal (‘equation solving’).

What is Unification?

- In a nutshell: the process of finding a substitution that makes two given terms equal ('equation solving').
- In detail:

What is Unification?

- In a nutshell: the process of finding a substitution that makes two given terms equal (‘equation solving’).
- In detail:
  - Terms have free variables

What is Unification?

- In a nutshell: the process of finding a substitution that makes two given terms equal ('equation solving').
- In detail:
  - Terms have free variables
  - A substitution is a map σ:Variables$\longrightarrow$Terms (write $\sigma = [t/x, s/y, \dots]$)

What is Unification?

- In a nutshell: the process of finding a substitution that makes two given terms equal ('equation solving').
- In detail:
  - Terms have free variables
  - A substitution is a map σ:Variables$\longrightarrow$Terms (write $\sigma = [t/x, s/y, \dots]$)
  - Applying σ to a term t means replacing each free variable v in t by $\sigma(v)$ — write $t\sigma$ for the result. Similarly: $\tau\sigma$, τ substitution

What is Unification?

- In a nutshell: the process of finding a substitution that makes two given terms equal ('equation solving').
- In detail:
  - Terms have free variables
  - A substitution is a map σ:Variables$\to$Terms (write $\sigma = [t/x, s/y, \dots]$)
  - Applying σ to a term t means replacing each free variable v in t by $\sigma(v)$ — write $t\sigma$ for the result. Similarly: $\tau\sigma$, τ substitution
  - A unifier of two terms s and t is a substitution σ such that $s\sigma = t\sigma$.

What is Unification?

- In a nutshell: the process of finding a substitution that makes two given terms equal (‘equation solving’).
- In detail:
  - Terms have free variables
  - A substitution is a map σ:Variables$\to$Terms (write $\sigma = [t/x, s/y, \dots]$)
  - Applying σ to a term t means replacing each free variable v in t by $\sigma(v)$ — write $t\sigma$ for the result. Similarly: $\tau\sigma$, τ substitution
  - A unifier of two terms s and t is a substitution σ such that $s\sigma = t\sigma$.
- Ooof.

We do this every day:

We do this every day:

Boolean terms:

We do this every day:

Boolean terms:

$$mother(x, y) \wedge mother(x, Paul) \implies sibling(y, Paul)$$

We do this every day:

Boolean terms:

$$mother(x, y) \wedge mother(x, Paul) \implies sibling(y, Paul)$$

$$mother(Mary, y) \wedge mother(Mary, z) \implies sibling(y, z)$$

We do this every day:

Boolean terms:

$$mother(x, y) \wedge mother(x, Paul) \implies sibling(y, Paul)$$

$$mother(Mary, y) \wedge mother(Mary, z) \implies sibling(y, z)$$

Unifier is $[x/Mary, z/Paul]$:

We do this every day:

Boolean terms:

$$mother(x, y) \wedge mother(x, Paul) \implies sibling(y, Paul)$$

$$mother(Mary, y) \wedge mother(Mary, z) \implies sibling(y, z)$$

Unifier is $[x/Mary, z/Paul]$:

$$mother(Mary, y) \wedge mother(Mary, Paul)$$
$$\implies sibling(y, Paul)$$

We do this every day:

Boolean terms:

$$mother(x, y) \land mother(x, Paul) \implies sibling(y, Paul)$$
$$mother(Mary, y) \land mother(Mary, z) \implies sibling(y, z)$$

Unifier is $[x/Mary, z/Paul]$:

$$mother(Mary, y) \land mother(Mary, Paul)$$
$$\implies sibling(y, Paul)$$

N.B.: $[Mary/x, Peter/y, Paul/z]$ is also a unifier!

This illustrates: substitution is specialization

One more example

$$t = (x + y) * z$$

$$s = u * (v + w)$$

One more example

$$t = (x + y) * z$$
$$s = u * (v + w)$$

Unifier is $\sigma = [(x + y)/u, (v + w)/z]$:

One more example

$$t = (x + y) * z$$

$$s = u * (v + w)$$

Unifier is $\sigma = [(x + y)/u, (v + w)/z]$:

$$(x + y) * (v + w)$$

One more example

$$t = (x + y) * z$$

$$s = u * (v + w)$$

Unifier is $\sigma = [(x + y)/u, (v + w)/z]$:

$$(x + y) * (v + w)$$

Use this in the right and left distributive laws to show

$$x * (v + w) + y * (v + w) = (x + y) * v + (x + y) * w$$

Unifiers may not exist

Unifiers may not exist

$$x + y \text{ and } x * y$$

do not admit a unifier.
(No matter what one substitutes, one always has a sum on the left and a product on the right.)

Unifiers may not exist

$$x + y \text{ and } x * y$$

do not admit a unifier.
(No matter what one substitutes, one always has a sum on the left and a product on the right.)
Neither do

$$x \text{ and } x + 1$$

(No matter what one substitutes, the right term is always larger.)

The most general unifier (mgu)

The most general unifier (mgu)

- Does ‘as little as possible’

The most general unifier (mgu)

- Does 'as little as possible'
- Formally:
  - σ is more general than τ if $\tau = \sigma\bar{\tau}$ for some $\bar{\tau}$;
  - A unifier of s and t is an mgu if it is more general than all other unifiers :-)

The most general unifier (mgu)

- Does 'as little as possible'
- Formally:
  - σ is more general than τ if $\tau = \sigma\bar{\tau}$ for some $\bar{\tau}$;
  - A unifier of s and t is an mgu if it is more general than all other unifiers :-)
- Above: σ is mgu

The most general unifier (mgu)

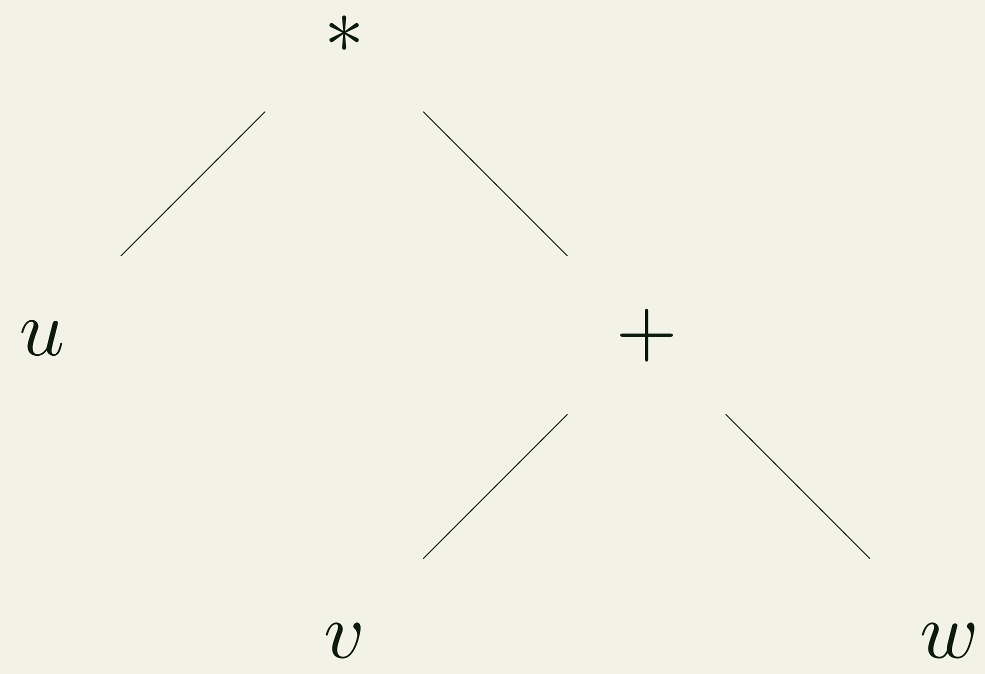
- Does 'as little as possible'
- Formally:
  - σ is more general than τ if $\tau = \sigma\bar{\tau}$ for some $\bar{\tau}$;
  - A unifier of s and t is an mgu if it is more general than all other unifiers :-)
- Above: σ is mgu
- $\tau = [(x + y)/u, (u * u + w)/z, (u * u)/v]$
  is also a unifier

The most general unifier (mgu)

- Does 'as little as possible'
- Formally:
  - σ is more general than τ if $\tau = \sigma\bar{\tau}$ for some $\bar{\tau}$;
  - A unifier of s and t is an mgu if it is more general than all other unifiers :-)
- Above: σ is mgu
- $\tau = [(x+y)/u, (u*u+w)/z, (u*u)/v]$
  is also a unifier
- $\tau = \sigma[u*u/v]$

Terms are Trees

E.g., $u * (v + w)$ is



Robinson's Algorithm

Robinson's Algorithm

Finds the mgu σ of terms s and t if a unifier exists:

Robinson's Algorithm

Finds the mgu σ of terms s and t if a unifier exists:

$\sigma := []$

while $s\sigma \neq t\sigma$ {

Find the first place in the tree where $s\sigma$ and $t\sigma$ have different subterms a, b

if none of a, b is a variable **return**("not unifiable")

else (w.l.o.g. a is a variable v)

if b contains v **return**("not unifiable")

else $\sigma := \sigma[b/v]$}

An example

Take $s = (x + y) * z$ and $t = w * x$.

An example

Take $s = (x + y) * z$ and $t = w * x$.

1. $x + y$ is different from and does not contain w:

$$\sigma := [(x + y)/w]$$

An example

Take $s = (x + y) * z$ and $t = w * x$.

1. $x + y$ is different from and does not contain w:

$$\sigma := [(x + y)/w]$$

2. z is different from and does not contain x:

$$\sigma := \sigma[x/z]$$

An example

Take $s = (x + y) * z$ and $t = w * x$.

1. $x + y$ is different from and does not contain w:

$$\sigma := [(x + y)/w]$$

2. z is different from and does not contain x:

$$\sigma := \sigma[x/z]$$

Result: $\sigma = [x/z, (x + y)/w]$, $s\sigma = (x + y) * x = t\sigma$.

N.B: $[z/x, (z + y)/w]$ would have done the job as well.

Why does this work?

Why does this work?

- The algorithm terminates: in each iteration, the number of free variables decreases by 1.

Why does this work?

- The algorithm terminates: in each iteration, the number of free variables decreases by 1.
- If the algorithm terminates successfully, it obviously returns a unifier (look at the while-condition); i.e. if there is no unifier, the result is correct.

Why does this work?

- The algorithm terminates: in each iteration, the number of free variables decreases by 1.
- If the algorithm terminates successfully, it obviously returns a unifier (look at the while-condition); i.e. if there is no unifier, the result is correct.
- If τ is a unifier, prove by induction that in each step, σ is more general than τ, so that

Why does this work?

- The algorithm terminates: in each iteration, the number of free variables decreases by 1.
- If the algorithm terminates successfully, it obviously returns a unifier (look at the while-condition); i.e. if there is no unifier, the result is correct.
- If τ is a unifier, prove by induction that in each step, σ is more general than τ, so that
  - $s\sigma$ and $t\sigma$ are always unifiable

Why does this work?

- The algorithm terminates: in each iteration, the number of free variables decreases by 1.
- If the algorithm terminates successfully, it obviously returns a unifier (look at the while-condition); i.e. if there is no unifier, the result is correct.
- If τ is a unifier, prove by induction that in each step, σ is more general than τ, so that
  - $s\sigma$ and $t\sigma$ are always unifiable
  - Thus, we are always in the **else** branches (cf. examples above)

Why does this work?

- The algorithm terminates: in each iteration, the number of free variables decreases by 1.
- If the algorithm terminates successfully, it obviously returns a unifier (look at the while-condition); i.e. if there is no unifier, the result is correct.
- If τ is a unifier, prove by induction that in each step, σ is more general than τ, so that
  - $s\sigma$ and $t\sigma$ are always unifiable
  - Thus, we are always in the **else** branches (cf. examples above)

- In particular, the algorithm does not fail and returns an mgu.

The inductive proof

The inductive proof

- Base case $\sigma = []$ is trivial.

The inductive proof

- Base case $\sigma = []$ is trivial.
- Assume $\tau = \sigma\bar{\tau}$ at the beginning of a step.

The inductive proof

- Base case $\sigma = []$ is trivial.
- Assume $\tau = \sigma\bar{\tau}$ at the beginning of a step.
- Let $\sigma' = \sigma[a/v]$ (i.e. 'the σ of the next step')
- Let $\bar{\tau}'$ be the same as $\bar{\tau}$ except that $v\bar{\tau}' = v$.

The inductive proof

- Base case $\sigma = []$ is trivial.
- Assume $\tau = \sigma\bar{\tau}$ at the beginning of a step.
- Let $\sigma' = \sigma[a/v]$ (i.e. 'the σ of the next step')
- Let $\bar{\tau}'$ be the same as $\bar{\tau}$ except that $v\bar{\tau}' = v$.
- Note: $v\bar{\tau} = a\bar{\tau}$.

The inductive proof

- Base case $\sigma = []$ is trivial.
- Assume $\tau = \sigma\bar{\tau}$ at the beginning of a step.
- Let $\sigma' = \sigma[a/v]$ (i.e. 'the σ of the next step')
- Let $\bar{\tau}'$ be the same as $\bar{\tau}$ except that $v\bar{\tau}' = v$.
- Note: $v\bar{\tau} = a\bar{\tau}$.
- Claim: $\tau = \sigma'\bar{\tau}'$.

The inductive proof

- Base case $\sigma = []$ is trivial.
- Assume $\tau = \sigma\bar{\tau}$ at the beginning of a step.
- Let $\sigma' = \sigma[a/v]$ (i.e. 'the σ of the next step')
- Let $\bar{\tau}'$ be the same as $\bar{\tau}$ except that $v\bar{\tau}' = v$.
- Note: $v\bar{\tau} = a\bar{\tau}$.
- Claim: $\tau = \sigma'\bar{\tau}'$.

Proof of the Claim

Proof of the Claim

$\sigma'\bar{\tau}'$

Proof of the Claim

$$\sigma'\bar{\tau}'$$

$$= \sigma[a/v]\bar{\tau}'$$

Proof of the Claim

$$
\begin{aligned}
&\sigma'\bar{\tau}' \\
&\quad = \sigma[a/v]\bar{\tau}' \\
&\quad = \sigma\bar{\tau}'[a\bar{\tau}'/v] && \text{(since } v\bar{\tau}' = v\text{)}
\end{aligned}
$$

Proof of the Claim

$$
\begin{aligned}
&\sigma'\bar{\tau}' \\
&\quad = \sigma[a/v]\bar{\tau}' \\
&\quad = \sigma\bar{\tau}'[a\bar{\tau}'/v] && \text{(since } v\bar{\tau}' = v\text{)} \\
&\quad = \sigma\bar{\tau}'[a\bar{\tau}/v] && (v \text{ not in } a)
\end{aligned}
$$

Proof of the Claim

$$
\begin{aligned}
&\sigma'\bar{\tau}' \\
&\quad = \sigma[a/v]\bar{\tau}' \\
&\quad = \sigma\bar{\tau}'[a\bar{\tau}'/v] && (\text{since } v\bar{\tau}' = v) \\
&\quad = \sigma\bar{\tau}'[a\bar{\tau}/v] && (v \text{ not in } a) \\
&\quad = \sigma\bar{\tau} && (\text{since } v\bar{\tau} = a\bar{\tau})
\end{aligned}
$$

Proof of the Claim

$$
\begin{aligned}
&\sigma'\bar{\tau}' \\
&\quad = \sigma[a/v]\bar{\tau}' \\
&\quad = \sigma\bar{\tau}'[a\bar{\tau}'/v] && \text{(since } v\bar{\tau}' = v\text{)} \\
&\quad = \sigma\bar{\tau}'[a\bar{\tau}/v] && (v \text{ not in } a) \\
&\quad = \sigma\bar{\tau} && \text{(since } v\bar{\tau} = a\bar{\tau}\text{)} \\
&\quad = \tau.
\end{aligned}
$$

Higher Order Unification

Higher Order Unification

- . . . is a lot harder.

Higher Order Unification

- . . . is a lot harder.
- The reason is that it concerns terms modulo certain equations, so that we get more unifiers

Higher Order Unification

- . . . is a lot harder.
- The reason is that it concerns terms modulo certain equations, so that we get more unifiers
- Specifically, we have to unify λ-terms modulo β-equality.

What was that again?

What was that again?

- Recall λ-notation from functional programming:

What was that again?

- Recall λ-notation from functional programming:

$$\lambda x.\, t$$

means 'the function that maps x to t' (where t may contain x as a free variable — as in $\lambda x.\, x + x$)

What was that again?

- Recall λ-notation from functional programming:

$$\lambda x.\, t$$

means 'the function that maps x to t' (where t may contain x as a free variable — as in $\lambda x.\, x + x$)

- β-equality is

$$(\lambda x.\, t)(s) = t[s/x]$$

What was that again?

- Recall λ-notation from functional programming:

$$\lambda x.\, t$$

  means 'the function that maps x to t' (where t may contain x as a free variable — as in $\lambda x.\, x + x$)
- β-equality is

$$(\lambda x.\, t)(s) = t[s/x]$$

- E.g. $(\lambda x.\, x + x)(3 * y) = 3 * y + 3 * y$.

Huet's Procedure

Huet's Procedure

Huet, G. P.: A unification algorithm for typed λ-calculus. *Theoret. Comput. Sci.* **1** (1975), 27–57.

Huet's Procedure

Huet, G. P.: A unification algorithm for typed λ-calculus. *Theoret. Comput. Sci.* **1** (1975), 27–57.

- Works by systematic 'guessing'

Huet's Procedure

Huet, G. P.: A unification algorithm for typed λ-calculus. *Theoret. Comput. Sci.* **1** (1975), 27–57.

- Works by systematic 'guessing'
- Is complete, i.e. finds all unifiers (unlike unification in Isabelle, which is even harder due to type variables)

Huet's Procedure

Huet, G. P.: A unification algorithm for typed λ-calculus. *Theoret. Comput. Sci.* **1** (1975), 27–57.

- Works by systematic 'guessing'
- Is complete, i.e. finds all unifiers (unlike unification in Isabelle, which is even harder due to type variables)
- May fail to terminate (i.e. may keep putting out unifiers)

How to guess unifiers

How to guess unifiers

Problem: Solve

$$?f(t_1, \ldots, t_m) = g(u_1, \ldots, u_n)$$

(From now on, unification variables are indicated by '?')

How to guess unifiers

Problem: Solve

$$?f(t_1, \dots, t_m) = g(u_1, \dots, u_n)$$

(From now on, unification variables are indicated by '?')

Two types of guesses for $?f$:

How to guess unifiers

Problem: Solve

$$?f(t_1, \ldots, t_m) = g(u_1, \ldots, u_n)$$

(From now on, unification variables are indicated by '?')

Two types of guesses for $?f$:

- Imitation: assume that $?f$ applies g to something.

How to guess unifiers

Problem: Solve

$$?f(t_1, \dots, t_m) = g(u_1, \dots, u_n)$$

(From now on, unification variables are indicated by '?')

Two types of guesses for $?f$:

- Imitation: assume that $?f$ applies g to something.
- Projection: assume that $?f$ applies one of the t_i to something (i.e. hope that g is hidden somewhere in t_i).

Imitation

Imitation

(Assume $m = 1$ for simplicity)

Imitation

(Assume $m = 1$ for simplicity)

- Guess $?f = \lambda x.\, g(?h_1(x), \ldots, ?h_n(x))$

Imitation

(Assume $m = 1$ for simplicity)

- Guess $?f = \lambda x.\, g(?h_1(x), \ldots, ?h_n(x))$
- Plug this into the equation and obtain (after β-reduction)

$$g(?h_1(t), \ldots, ?h_n(t)) = g(u_1, \ldots, u_n)$$

Imitation

(Assume $m = 1$ for simplicity)

- Guess $?f = \lambda x.\, g(?h_1(x), \ldots, ?h_n(x))$
- Plug this into the equation and obtain (after β-reduction)

$$g(?h_1(t), \ldots, ?h_n(t)) = g(u_1, \ldots, u_n)$$

- Next step: solve

$$?h_i(t) = u_i, i = 1, \ldots, n$$

Projection

Projection

(Works if t has the same result type as g)

Projection

(Works if t has the same result type as g)

- Guess $?f = \lambda x.\, x(?h_1(x), \ldots, ?h_k(x))$, where k is the number of arguments for t

Projection

(Works if t has the same result type as g)

- Guess $?f = \lambda x.\, x(?h_1(x), \ldots, ?h_k(x))$, where k is the number of arguments for t
- Plug in and β-reduce:

$$t(?h_1(t), \ldots, ?h_k(t)) = g(u_1, \ldots, u_n)$$

Projection

(Works if t has the same result type as g)

- Guess $?f = \lambda x.\, x(?h_1(x), \ldots, ?h_k(x))$, where k is the number of arguments for t
- Plug in and β-reduce:

$$t(?h_1(t), \ldots, ?h_k(t)) = g(u_1, \ldots, u_n)$$

- this is a ‘simpler’ problem, since the unknowns have been pushed ‘further down’ in the term.

An Example

An Example

Solve $?f(a) = a + a$.

An Example

Solve $?f(a) = a + a$.

- Imitation: $?f = \lambda x.\, ?h_1(x) + ?h_2(x)$, have to solve $?h_i(a) = a$, $i = 1, 2$, next.

An Example

Solve $?f(a) = a + a$.

- Imitation: $?f = \lambda x.\,?h_1(x)+?h_2(x)$,
  have to solve $?h_i(a) = a$, $i = 1, 2$, next.
- Imitation: $?h_i = \lambda x.\,a$.

An Example

Solve $?f(a) = a + a$.

- Imitation: $?f = \lambda x.\, ?h_1(x) + ?h_2(x)$,
  have to solve $?h_i(a) = a$, $i = 1, 2$, next.
- Imitation: $?h_i = \lambda x.\, a$.
- Alternatively, projection (a takes 0 arguments!):

$$?h_i = \lambda x.\, x,$$

  then have to 'solve' $a = a$.

Example (cont'd)

Example (cont'd)

- Thus: four solutions

$$\lambda x.\, x + x, \lambda x.\, x + a, \lambda x.\, a + x, \lambda x.\, a + a$$

for $?f$.

Example (cont'd)

- Thus: four solutions

$$\lambda x.\, x + x, \lambda x.\, x + a, \lambda x.\, a + x, \lambda x.\, a + a$$

  for $?f$.

- None of these is more general than another, since we cannot substitute for bound variables.

Example (cont'd)

- Thus: four solutions

$$\lambda x.\, x + x, \lambda x.\, x + a, \lambda x.\, a + x, \lambda x.\, a + a$$

  for $?f$.

- None of these is more general than another, since we cannot substitute for bound variables.

$\boxed{\text{Who on earth would believe } \textit{that}\text{?}}$