Verifying the Unification Algorithm in HOL

Our goal is to verify the standard algorithm for unification before the end of semester.

Description of the Algorithm

Definition: We assume an infinite set of variables $\mathcal{V}$.

We also assume a set of functions $\mathcal{F}$. Each function $f \in \mathcal{F}$ has an arity $\#f$ attached to it, for which $\#f \geq 0$. A function c with $\#c = 0$ is called constant.

Definition: The set of terms is recursively defined as follows:

- A variable $V \in \mathcal{V}$ is a term.
- If $t_1, \ldots, t_n$ are terms, $f \in \mathcal{F}$ has arity n, then $f(t_1, \ldots, t_n)$ is a term.

Definition: A term is ground if it contains no variables.

Substitutions/Unification

Definition: A substitution is a finite set of assignments of form $\{V_1 := t_1, \ldots, V_n := t_n\}$. The V_i are variables, the t_i are terms. It must be the case that $V_i = V_j$ implies $t_i = t_j$.

The application of a substitution Θ on a term t, $t \cdot \Theta$ is recursively defined as follows:

- If V equals one of the V_i then $V \cdot \Theta = t_i$.
- If V does not equal any of the V_i, then $V \cdot \Theta = V$.
- $f(t_1, \ldots, t_n) \cdot \Theta = f(t_1 \cdot \Theta, \ldots, t_n \cdot \Theta)$.

(We usually will not write the $\cdot$ in applications)

Composition

Definition: Let Θ_1 and Θ_2 be substitutions. The composition $\Theta_1 \cdot \Theta_2$ of Θ_1 and Θ_2 is defined as

$$\{V := t \mid t = (V \cdot \Theta_1) \cdot \Theta_2 \text{ and } V \neq t\}.$$

Theorem: For each term t, and substitutions Θ_1, Θ_2,

$$t \cdot (\Theta_1 \cdot \Theta_2) = (t \cdot \Theta_1) \cdot \Theta_2.$$

proof: We first prove the theorem for variables. Let V be a variable. If $(V \cdot \Theta_1) \cdot \Theta_2 \neq V$, then $\Theta_1 \cdot \Theta_2$ contains the assignment $V := (V \cdot \Theta_1) \cdot \Theta_2$, so that $V \cdot (\Theta_1 \cdot \Theta_2) = (V \cdot \Theta_1) \cdot \Theta_2$.

If $(V \cdot \Theta_1) \cdot \Theta_2 = V$, then there is no assignment $V := t$ in $\Theta_1 \cdot \Theta_2$. Hence $V \cdot (\Theta_1 \cdot \Theta_2) = V$.

In both cases, $V \cdot (\Theta_1 \cdot \Theta_2) = (V \cdot \Theta_1) \cdot \Theta_2$.

Composition (2)

For non-variable terms, we apply induction. Assume the theorem holds for all subterms of $f(t_1, \ldots, t_n)$. Then

$$f(t_1, \ldots, t_n) \cdot (\Theta_1 \cdot \Theta_2) = f(\ t_1 \cdot (\Theta_1 \cdot \Theta_2), \ldots, t_n \cdot (\Theta_1 \cdot \Theta_2)\ ).$$

By induction, this is equal to $f(\ (t_1 \cdot \Theta_1) \cdot \Theta_2, \ldots, (t_n \cdot \Theta_1) \cdot \Theta_2) =$

$$f(\ (t_1 \cdot \Theta_1), \ldots, (t_n \cdot \Theta_1)\ ) \cdot \Theta_2 = (f(\ t_1, \ldots, t_n) \cdot \Theta_1\ ) \cdot \Theta_2.$$

Unification

Definition: Let $\mathcal{E} = [\, t_1 \equiv u_1, \ldots, t_n \equiv u_n \,]$ be a system of equations between terms.

A unifier of $\mathcal{E}$ is a substitution Θ s.t. $t_1\Theta = u_1\Theta, \ldots, t_n\Theta = u_n\Theta$.

A most general unifier is a unifier Θ, such that for every unifier Θ', there exists a substitution Σ, s.t. $\Theta' = \Theta \cdot \Sigma$.

A unifier of two terms t_1, t_2 is a unifier of the system of equation $[t_1 \equiv t_2]$. Similarly, a most general unifier of t_1, t_2 is a most general unifier of $[t_1 \equiv t_2]$.

Unification (2)

The terms $f(X, 1)$ and $f(0, Y)$ have unifier $\{X := 0,\ Y := 0\}$. This is also the most general unifier.

The terms $f(X, s(X))$ and $f(Y, Z)$ have unifier
$\{X := 0,\ Y := 0,\ Z := s(0)\}$. Is this a most general unifier?

Do $f(X, Y)$ and $f(a, Z)$ have a unifier?

And $f(X, Y)$ and $g(X, Y)$?

What about $f(X, Y)$ and $f(a, X)$?

And $f(X, s(X))$ and $f(s(Y), Y)$?

Most General Unifiers

G.A. Robinson showed the following (1965)

- If a system of equations has a unifier, then it has a most general unifier.
- There exists an algorithm that decides if a system of equations has a most general unifier. It also computes the most general unifier.

Algorithm for computing the MGU

The algorithm is called with a system of equations $\mathcal{E}$. It returns either a most general unifier of $\mathcal{E}$, or an error, in case no unifier exists. We represent the error by $\perp$.

Description of the Unification Algorithm

- $$\mathsf{mgu}(\mathcal{E} \cup [\ f(t_1, \ldots, t_n) \equiv f(u_1, \ldots, u_n)\ ]\ )$$

  returns

  $$\mathsf{mgu}(\mathcal{E} \cup [\ t_1 \equiv u_1, \cdots, t_n \equiv u_n\ ]\ ).$$

- If $f \neq g$, then

  $$\mathsf{mgu}(\mathcal{E} \cup [\ f(t_1, \ldots, t_n) \equiv g(u_1, \ldots, u_m)\ ]\ )$$

  returns $\perp$.

- $$\mathsf{mgu}(\mathcal{E} \cup [\ t \equiv t\ ]\ )$$

  returns

  $$\mathsf{mgu}(\mathcal{E}).$$

- When x is a variable that does not occur in t, then

$$\mathsf{mgu}(\mathcal{E} \cup [\ x \equiv t\ ]\ )$$

  returns

$$\{x := t\} \cdot \mathsf{mgu}(\mathcal{E}).$$

- When x is a variable that does occur in t, but is not equal to t, then

$$\mathsf{mgu}(\mathcal{E} \cup [\ x \equiv t\ ]\ ) = \bot.$$

- $\mathsf{mgu}([\ ]) = \{\ \}$.

Datatypes in HOL

We first need to decide about the representation of terms. We have to make the following choices:

- Do we want full terms (like $f(t_1, \ldots, t_n)$) or will we use Currying? Currying is certainly easier, but there exist Curried terms that do not correspond to a usual term (for example $x \cdot t$)
- Do we want a fixed arity attached to each symbol, or do we allow the same symbol to occur with different arities. The latter is easier, because then we don't need to check when building a term. Fixed arities, would make the constructors partial, which is extremely unpleasant.

Datatypes in HOL (2)

I propose: No Currying, no fixed arities.

That would give rise to the following coinductive definition:

$$\mathsf{term} = \mathsf{Variable\ num} \mid \mathsf{Application\ num\ termlist};$$

$$\mathsf{termlist} = \mathsf{Empty} \mid \mathsf{Cons\ term\ termlist};$$

(Type this in HOL, and see what a beast of an induction/recursion principle you get)

Datatypes in HOL (3)

As for substitutions, there are quite some choices to make:

- Is it essential that they are finite? If yes, then some inductive data type is appropriate. If not, a substitution can be just a function from variables to terms.
- If we choose functions, we should consider making variables a separate type. We should worry about how we will construct the substitutions that we need. We will have problems stating the theorem about composition of substitutions.
- If we choose a finite representation, we should invent some inductive type. (or simply use HOL lists of HOL pairs)

  There is a problem that not every list will represent a substitution. (Due to conflicting assignments) and that different lists will represent the same substitution.

Datatypes in HOL (4)

I propose: An inductive representation, based on HOL lists.

In case there are conflicting assignments, the first assignment counts.

I think we don't care that the same substitution has many representations. We don't need uniqueness of representation.

If we think that unique representation is important, we can use define_quotient_type.

Datatypes in HOL (5)

I think that the system of equations can be represented by a list of pairs of terms.

Dealing with $\perp$

When the mgu function does not construct a unifier, it returns $\perp$.

I propose to introduce a new inductive type for partial substitutions as follows:

$$\mathsf{partialsubst} = \mathsf{Ok\ subst} \mid \mathsf{Error}.$$

Non-determinism

- When $|\mathcal{E}| \geq 2$, the specification does not specify which equation should be expanded.
- If the expanded equation has form $f(t_1, \ldots, t_n) \equiv f(t_1, \ldots, t_n)$, we can ether drop the equation completely, or replace it by $t_1 \equiv t_1, \ldots, t_n \equiv t_n$.

Non-determinism (2)

We can either

1. Make the algorithm deterministic in some way. (Always expand first/smallest equation, or order the rules)
2. Cope with the non-determinism in our verification.

As for (1), we have to make choices. Whatever choice we make, it is unlikely that we will make the implementer happy.

(He may want to make the choices different. He may even be unable to make all choices, because it can be the compiler/run time environment that eventually decides)

We conclude that:

All serious specifications are non-deterministic, and we have to deal with it.

Non-determinism (3)

I propose to

1. Leave the non-determinism in the choice of the equation.
2. Restrict the removal of trivial equations $t \equiv t$ to the case where t is a variable.

Specification of the Unification Algorithm

We define a choice function $S(\mathcal{E})$ that decides which equation will be expanded.

$$\mathsf{nonempty}(\mathcal{E}) \rightarrow S(\mathcal{E}) \in \mathcal{E}.$$

For each choice function S, $\mathsf{mgu}(S)$ is the smallest relation (over (Type of sets of equations) $\times$ (Type of partial substitutions)) that satisfies the following axioms:

- if $\mathcal{E}$ is nonempty, $S(\mathcal{E})$ has form $f(t_1, \ldots, t_n) \equiv f(u_1, \ldots, u_n)$, and $\mathsf{mgu}(S, [\ t_1 \equiv u_1, \ldots, t_n \equiv u_n\ ] \cup (\mathcal{E} \backslash S(\mathcal{E})), r)$, then $\mathsf{mgu}(S, \mathcal{E}, r)$.
- if $\mathcal{E}$ is nonempty, $S(\mathcal{E})$ has form $f(t_1, \ldots, t_n) \equiv g(u_1, \ldots, u_m)$, with $f \neq g$, then $\mathsf{mgu}(S, \mathcal{E}, \bot)$.
- if $\mathcal{E}$ is nonempty, $S(\mathcal{E})$ has form $x \equiv x$, with x a variable, and $\mathsf{mgu}(S, (\mathcal{E} \backslash S(\mathcal{E})), r)$, then $\mathsf{mgu}(S, \mathcal{E}, r)$.

- if $\mathcal{E}$ is nonempty, $S(\mathcal{E})$ has form $x \equiv t$, where x is a variable that does not occur in t, and $\mathsf{mgu}(S, (\mathcal{E} \backslash S(\mathcal{E}))\{x := t\}, r)$, then $\mathsf{mgu}(S, \mathcal{E}, \{x := t\} \cdot r)$.
- if $\mathcal{E}$ is nonempty, $S(\mathcal{E})$ has form $x \equiv t$, $\quad x$ occurs in t, and $x \neq t$, then $\mathsf{mgu}(S, \mathcal{E}, \bot)$.
- $\mathsf{mgu}(S, [\,], \{\,\})$.

Transfinite Induction/Recursion

We have seen data type induction. Every recursive data type comes with an induction and a recursion axiom which follow the tree structure of the data type.

The unification algorithm does not fit into this pattern.

Its termination is based on the fact that it either deletes a variable, or decreases the total weight of its equations while not introducing a new variable.

Transfinite Induction/Recursion (2)

Definition: Let $<$ be a relation on some type S.

$<$ is called well-founded if for all predicates P on S :

$$\forall s\ (\forall s'\ s' < s \rightarrow P(s')) \rightarrow P(s)$$

implies

$$\forall s\ P(s).$$

$<$ on the natural numbers is such a relation.

1. Is $<$ on Z WF?
2. Is the dictionary ordering on all strings WF?

Transfinite Induction/Recursion (3)

We will need to prove the following theorems (or maybe they are already proven in HOL)

Theorem WF1: Let S be a set. Let f be a function from S to N. Define $s_1 \prec s_2$ iff $f(s_1) < f(s_2)$. Then $\prec$ is WF on S.

Theorem WF2: Let S_1, S_2 be sets. Let $\prec_1$ be WF on S_1, let $\prec_2$ be WF on S_2.

Define $\prec$ from $(s_1, s_2) \prec (t_1, t_2)$ iff $s_1 \prec_1 t_1$, or $s_1 = t_1$ and $s_2 \prec_2 t_2$.

Then $\prec$ is WF on $S_1 \times S_2$.

Question: For which f do we need theorem WF1? Where will WF2 be used?