Unification in Maude

Steven Eker

Protocol eXchange Seminar

Unification

- Unification is essentially solving equations in an abstract setting.
- Given a signature Σ, variables X and terms $t_1, t_2 \in T(\Sigma)$ we want to find substitutions $\sigma : X \to T(W)$, where W is some fresh set of variables such that

$$t_1\sigma = t_2\sigma$$

- A unifier σ_1 is *more general* that a unifier σ_2 if there is a substitution υ such that $\sigma_1\upsilon = \sigma_2$. σ_2 is an *instance* of σ_1.
- For most applications we want a *complete set* C of unifiers - every unifer of $t_1 =_? t_2$ would be an instance of some unifer in C.
- Computing a minimal C (no redundant unifiers) would be nice but can be expensive.

Unification in Maude

Unification in Maude is complicated by two factors:

1. Equational theories: rather than equality between substituted terms we want equality between their congruence classes:

$$[t_1\sigma]_E = [t_2\sigma]_E$$

   where E is a set of axioms supported by Maude.

2. Membership equational logic: assignments to variables in unifiers must respect the sorts of the variables. In fact we punt on on MEL (undecideable in general) and just consider order sort algebra.

Either factor on its own:

1. Results in non-singleton minimal complete sets of unifiers.
2. Bumps the complexity of the decision problem (is there a unifer?) from linear time to NP-hard.

Order-sorted free theory unification

Given a two free-theory terms t_1, t_2 from a pre-regular order-sorted signature, deciding where thay are unifable, $t_1 =_? t_2$, is NP-complete.

To see that the problem is in NP we note that given a putative unifier σ:

1. $t_1\sigma = t_2\sigma$ can be checked in linear time substitution.
2. The sort of a term t can be computed in polynomial time by proceeding bottom up and at each operator occurence, looking at all the operator declarations that have a sufficiently large coarity and taking the glb of their arities (unique by preregularity).
3. Thus for each assignment $X \leftarrow t$ in σ we can check in polynomial time that the sort of t is less or equal to the sort of X.

NP-hardness

The basic idea is to encode satisfiability using the following signature:

```
fmod ENCODE is
  sorts True False Value .
  subsort True False < Value .
  op true : -> True .
  op false : -> False .
  op not : True -> False .
  op not : False -> True .
  op and : True True -> True .
  op and : True False -> False .
  op and : False True -> False .
  op and : False False -> False .
  op or : True True -> True .
  op or : True False -> True .
  op or : False True -> True .
  op or : False False -> False .
endfm
```

NP-hardness 2

Let F be any propositional formula over propositions $p_1, \ldots, p_n$. Translate F into a term t over the signature `ENCODE`:

- Proposition p_i is replaced by a variable X_i of sort `Value`.
- Each propositional connective is replaced by the corresponding operator.

Let Y be a variable of sort `True`. Then the order sorted unification problem $Y =_? t$ has a solution iff F is satisfiable, since Y and t are unifiable iff there is an assignment of sorts to $X_1, \ldots, X_n$ such that t has sort `True` and the sort declarations exactly mirror the semantics of the propositional connectives.

Implementation

Unsorted unification in the free theory is well understood. For order-sorted unification we:

- Find the unique unsorted unifier using a standard algorithm.
- Search for all ways of giving sorts to free variables in the unifier such that:
  - Each free variable recieves a sort that is less or equal to its original sort.
  - The term assigned to each bound variable has a sort less or equal than the sort of that variable.
- For each such assignment of sorts to free variables, compute an order-sorted unifier by replacing the free variables in the unsorted unifier by fresh variables of the appropriate sort.

Top-down constraint propagation

Consider an assignment $Y : s \leftarrow t$ in the unsorted unifier. That gives us an initial constraint that $sort(t) \leq s$.

- Suppose $t = f(t_1, \ldots, t_n)$.
- A declaration $f : s_1 \ldots s_n \rightarrow s'$ of largest arity such that $s' < s$ gives us a conjunction of new constraints
  $sort(t_1) \leq s_1 \wedge \cdots \wedge sort(t_n) \leq s_n$.
- Multiple such declarations leads to a disjunction of conjunctions.
- The non-existence of a declaration for f with coarity $\leq s$ leads to failure.

The constraints can be percolated down until ultimately we arrive at a nesting of conjunctions and disjunctions of constraints on the sorts the free variables for each assignment on the unsorted unifier.

Top-down constraint propagation 2

- These expressions can be conjuncted together and the result placed in DNF where each conjunction is a conjunction of constraints the free variables.
- Each free variable already has a constraint on it's sort given by it original declared sort.
- Multiple constraints on the same variable can be resolved by a glb computation, possibly producing several alternatives - or none at all.
- Thus each of the conjuncts may give rise to 0 or more order-sorted unifiers.

This approach can introduce redundant unifiers.

Top-down constraint propagation 3

Consider the unification problem

$$f(A : NzNat, B : NzNat) =_? f(X : Nat + Y : Nat, Y : Nat + Z : Nat)$$

in the signature

```
fmod NAT is
  sorts NzNat Nat .
  subsort NzNat < Nat .
  op _+_ : Nat Nat -> Nat .
  op _+_ : NzNat Nat -> NzNat .
  op _+_ : Nat NzNat -> NzNat .
  op 0 : -> Nat .
  op s : Nat -> NzNat .
  op f : Nat Nat -> Nat .
endfm
```

The unsorted unifier is

$$A : NzNat \leftarrow X : Nat + Y : Nat$$

$$B : NzNat \leftarrow Y : Nat + Z : Nat$$

Top-down constraint propagation 4

Resolving $sort(X : Nat + Y : Nat) \leq NzNat$ we get:

$$sort(X) \leq NzNat \wedge sort(Y) \leq Nat \vee sort(X) \leq Nat \wedge sort(Y) \leq NzNat$$

Similarly resolving $sort(Y : Nat + Z : Nat) \leq NzNat$ we get:

$$sort(Y) \leq NzNat \wedge sort(Z) \leq Nat \vee sort(Y) \leq Nat \wedge sort(Z) \leq NzNat$$

Conjuncting these two constraints together and distributing out we get 4 conjunctions:

$$sort(X) \leq NzNat \wedge sort(Y) \leq Nat \wedge sort(Y) \leq NzNat \wedge sort(Z) \leq Nat$$

$$sort(X) \leq NzNat \wedge sort(Y) \leq Nat \wedge sort(Y) \leq Nat \wedge sort(Z) \leq NzNat$$

$$sort(X) \leq Nat \wedge sort(Y) \leq NzNat \wedge sort(Y) \leq NzNat \wedge sort(Z) \leq Nat$$

$$sort(X) \leq Nat \wedge sort(Y) \leq NzNat \wedge sort(Y) \leq Nat \wedge sort(Z) \leq NzNat$$

Top-down constraint propagation 5

Resolving the multiple constraints on Y we get

$$sort(X) = NzNat \wedge sort(Y) = NzNat \wedge sort(Z) = Nat$$

$$sort(X) = NzNat \wedge sort(Y) = Nat \wedge sort(Z) = NzNat$$

$$sort(X) = Nat \wedge sort(Y) = NzNat \wedge sort(Z) = Nat$$

$$sort(X) = Nat \wedge sort(Y) = NzNat \wedge sort(Z) = NzNat$$

However the 1st and 4th alternatives are subsumed by the 3rd alternative so there are only 2 most general unifiers corresponding to the 2nd and 3rd alternatives.

Bottom-up symbolic sort computation

Boolean Decision Diagrams (BDDs) can be viewed as compressed truth tables.

- For random Boolean functions they are less compact than truth tables.
- Boolean functions of interest usually have a lot of structure — BDDs can be much more compact than truth tables — perhaps even exponentially so if we lucky.
- Many useful operations (combination by arbitrary Boolean connective, composition, univeral and existential elimination of variables, substitution of variables) can be performed efficiently.
- Many implementations available as open source libraries.

Bottom-up symbolic sort computation 2

- Represent sorts symbolically using BDDs.
- If a kind has n user sorts we need to represent $n+1$ values and thus need $\lceil(\log_2(n+1)\rceil$ BDD variables to represent a sort from this kind.
- For each operator we precompute the vector of BDDs that encodes its sort function — this can be done reasonably efficiently if (as in Maude) an ordered decision diagram for computing this sort function has already be generated.
- For each sort s we precompute the BDD for the relation $lt_s(s') = s' \leq s$.
- For each kind k, we precompute the BDD for the relation $s > s'$.

Bottom-up symbolic sort computation 3

- We represent the sort of each free variable $X : s$ by a vector of BDD variables.
- For each assignment $Y : s' \leftarrow t$ in the unsorted unifer we can bottom-up symbolically compute the sort of t as a vector of BDDs by using BDD vector composition.
- We can then compute a BDD for the constraint that $sort(t) \leq s'$.
- For each free variable $X : s$ we can also compute a BDD for the constraint that $sort(X) \leq s$.
- Conjuncting together both kinds of constraints we arrive a BDD encoding a relation osu on the sorts of the free variables that exactly corresponds to sort assignments yielding order-sorted unifiers.

Bottom-up symbolic sort computation 4

- Often there will be a huge number of such sort assignments and we are only interested in those yielding most general unifiers.
- Want to compute a relation mgu that encodes these sort assignments.
- Let vector of kinds on which osu is defined be $(K_1, \ldots, K_n)$.
- The sorts in each kind are partially ordered and thus the pointwise extension of these orderings gives a partial ordering on $K_1 \times \cdots \times K_n$.
- For $v, v' \in K_1 \times \cdots \times K_n$, if $osu(v)$ and $v' \leq v$ then $osu(v')$.
- Furthermore if $v \neq v'$ then $v' < v$ by antisymmmetry and v' cannot yield a most general unifier.

Bottom-up symbolic sort computation 5

For $v' \in K_1 \times \cdots \times K_n$, if $osu(v')$ but v' does not yield a most general unifier then the must be some unifier, different from v' that is more general, given by a sort vector v.

Clearly $v \geq v'$ since the more general unifier must be at least as general as that given be v' on every variable position. And we can't have $v = v'$ since we need a distinct unifier, so $v > v'$.

Thus to compute the relation mgu that encodes exactly the most general unifiers we want to compute

$$mgu(v) = osu(v) \wedge \neg(\exists v'.[v' > v \wedge osu(v')])$$

Computation can be expensive since it doubles the number of BDD variables in the intermediate expressions.

Bottom-up symbolic sort computation 6

We note that if $v' > v$ and $osu(v')$ then

1. for all v'' such that $v' > v''$, we have $osu(v'')$; and
2. there must exist some index j at which the sort in v' is great than the sort in v.

Thus there exists a vector u which is equal to v on all components except j and greater than v on component j, and $osu(u)$.

Thus to assert $\neg(\exists v'.[v' > v \wedge osu(v')])$ we form a conjunction of assertions, one for each of the possible indices j:

$$\text{for } j = 1, \ldots, n.[\neg(\exists s.[s > v_j \wedge osu(v_1, \ldots, v_{j-1}, s, v_{j+1}, \ldots, v_n)])]$$

Here we existentially quantify over sorts rather than vectors of sorts so we use fewer additional BDD variables in the intermediate expressions.

Bottom-up symbolic sort computation 7

Having computed a BDD that encodes mgu we recover the sort assignments to free variables that result in most general order-sorted unifiers.

- We trace the from the root of the BDD to the true terminal.
- Each such path corresponds to one or more sort assignments, given by the labels on arcs taken.
- Where some BDD variable is not mentioned on a path, both the true and false values must be used, giving rise to multiple sort assignments.

Maude Implementation

- The unify command has the syntax:

$$\texttt{unify}\ [\langle limit \rangle]\ \texttt{in}\ \langle module \rangle : \langle term \rangle =?\langle term \rangle .$$

- $[\langle limit \rangle]$ and $\texttt{in}\ \langle module$ parts are optional.
- Terms may only contain free function symbols.
- Unifiers are returned as subsitutions where all variables in the original problem are mappped to terms containing fresh variables with base names #1, #2, #3.
- The command

$$\texttt{cont}\ \langle limit \rangle .$$

  can be used to look for further unifiers if a limit was originally given.

Maude Implementation 2

- The unify command is reflected by the descent function:

```
op metaUnify : Module Term Term Nat ~> Substitution?
      [special (...)] .
```

- As with metaMatch(), the last argument controls which solution is returned.
- When there are no further unifiers

```
op noMatch : -> Substitution? [ctor] .
```

  is returned.

Next steps

- Non-algebraic data types (Strings, Floats, Qids).
- Iter theory (in order to support number hierarchy).
- Commutative and AC theories.

In the AC case, sort information can be pushed into the unsorted unification algorithm by computing bounds on what a given variable can take and using it to prune the Diophantive basis and choice of subsets drawn from the basis.