Program Verification: Lecture 7

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

Overlaps as Unification Problems

We reduced confluence (under the termination assumption) to joinability of context-free nested simplifications with overlap. But note that we can have a context-free overlap situation with equations $u = v$ and $u' = v'$ (again, with disjoint variables) if and only if there is a nonvariable position p in u and a substitution θ such that,

$$(\dagger) \quad u_p\theta = u'\theta.$$

Therefore, finding all possible context-free nested simplifications with overlap can be reduced to finding, for all pairs of equations $u = v$ and $u' = v'$ in E and all nonvariable positions p in u, all solutions to $(\dagger)$. Problems of the form $(\dagger)$ are called unification problems.

Unification

In general, the unification problem consists in, given terms t and t' whose sorts are in the same connected component, finding a substitution θ that makes them equal, so that we have identical terms, $t\theta = t'\theta$. The substitution θ is then called a unifier of t and t'.

Under very reasonable conditions on Σ, such as finiteness , this problem is decidable in a very strong sense. Namely, we can effectively find a finite set of unifiers $\{\theta_1, \ldots \theta_n\}$, that are the most general possible, in the sense that for any other substitution $\mu : vars(t = t') \longrightarrow T_\Sigma(V)$ such that $t\mu = t'\mu$, we can always find a θ_i, say, $\theta_i : vars(t = t') \longrightarrow T_\Sigma(X)$, and a substitution $\rho : X \longrightarrow T_\Sigma(V)$ such that for each $x \in vars(t = t')$ we have $x\mu = x\theta_i\,\rho$.

B-**Unification**

The standard unification problem is to try to unify two terms. But we have already encountered situations, such as the relation $\longrightarrow_{E/B}$, in which it is very useful to deal not with terms, but with equivalence classes of terms modulo some equational axioms B.

Therefore, it is natural, given a set of equational axioms B, such as the associativity, commutativity, and identity of some operators, to generalize the unification problem to the following B-unification problem: given an equation $t = t'$ are there substitutions θ such that

$$t\theta =_B t'\theta.$$

B-Unification (II)

For B any combination of associativity, commutativity, and identity axioms, there are known algorithms that can find a family of most general unifiers for each given unification problem $t = t'$. However, for the case of associativity alone, or of associativity and identity alone, this family of most general unifiers may be infinite.

In particular, for Σ a finite signature, if we choose B to be any combination of associativity, commutativity and identity axioms for different (subsort-overloaded) binary operators in Σ, except associativity without commutativity, there is indeed an algorithm that, given an B-unification problem, either declares the problem unsolvable, or finds a finite set of most general unifiers solving it. Such a B-unification algorithm is used by the Church-Rosser Checker.

More on Unification

So far we have said nothing about unification algorithms, that can effectively find a set of most general unifiers or declare the corresponding problem unsolvable.

Unification is indeed a vast research area, and the more we can do in this course is to give a flavor for the key ideas. This can be done quite well by considering the simplest version of the unification problem, for which, if the given equation has a solution, then it has a unique most general unifier.

More on Unification (II)

This simplest version is the case of a sensible many-sorted signature Σ without ad-hoc overloading.

The key idea of a unification algorithm is to transform the original equation we want to solve into a set of equations equivalent to the original equation, in the sense that both sets have the same solutions.

We then stop either with failure, or with a set of equations in solved form, that is, equations having the shape, $\{x_1 = t_1, \ldots, x_n = t_n\}$. But this is just another garb for a substitution $\theta = \{(x_1, t_1), \ldots, (x_n, t_n)\}$.

The Unification Algorithm

We can describe the unification algorithm, à la Martelli-Montanari, as a set of inference rules, that transform a set of equations E into another set of equations that is equivalent to it from the solvability point of view, or into the constant **failure**. The following inference rules assume that the equality symbol is commutative and use a global set V of variables:

- **Delete:**

$$\frac{\{E,\ t = t\}}{\{E\}}$$

- **Decompose:**

$$\frac{\{E,\ f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)\}}{\{E,\ t_1 = t'_1, \ldots, t_n = t'_n)\}}$$

The Unification Algorithm (II)

- **Conflict:**

$$\frac{\{E,\ f(t_1,\ldots,t_n) = g(t'_1,\ldots,t'_m)\}}{\textbf{failure}}$$

  if $f \neq g$

- **Coalesce:**

$$\frac{\{E,\ x = y\}}{\{E(x/y),\ x = y\}}$$

  if $x, y \in \text{vars}(E),\ x \neq y$

- **Check:**

$$\frac{\{E,\ x = t\}}{\textbf{failure}}$$

  if $x \in \text{vars}(t),\ x \neq t$

The Unification Algorithm (III)

- **Eliminate:**
$$\frac{\{E,\ x = t\}}{\{E(x/t),\ x = t\}}$$
if $x \notin \mathsf{vars}(t),\ t \notin V,\ x \in \mathsf{vars}(E)$.

We can illustrate the use of these rules by finding the most general unifier for a relatively simple, yet nontrivial, unification problem, namely, solving the equation,

$$f(g(x, h(y)), z) = f(z, g(k(u), v))$$

for which the above rules give us the following transformations:

$\{f(g(x, h(y)), z) = f(z, g(k(u), v))\} \longrightarrow$ (**Decompose**)

The Unification Algorithm (IV)

$\{g(x, h(y)) = z,\ z = g(k(u), v)\} \longrightarrow$ (**Eliminate**)

$\{g(x, h(y)) = g(k(u), v),\ z = g(k(u), v)\} \longrightarrow$ (**Decompose**)

$\{x = k(u),\ v = h(y),\ z = g(k(u), v)\} \longrightarrow$ (**Eliminate**)

$\{x = k(u),\ v = h(y),\ z = g(k(u), h(y))\}$,

which is the desired most general unifier, yielding the identity,

$$f(g(k(u), h(y)), g(k(u), h(y))) = f(g(k(u), h(y)), g(k(u), h(y))).$$

Unification Modulo Commutativity

To illustrate the case of B-unification in an unsorted signature Σ, let us assume that $B = Comm$ is a collection of commutativity axioms for some binary symbols $\Sigma_{comm} \subseteq \Sigma$, so that we have The inference rules for unification modulo commutativity are:

- **Delete:**

$$\frac{\{E,\ t = t\}}{\{E\}}$$

- **Decompose:** $(f \in (\Sigma - \Sigma_{comm}))$

$$\frac{\{E,\ f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)\}}{\{E,\ t_1 = t'_1, \ldots, t_n = t'_n)\}}$$

Unification Modulo Commutativity (II)

- **Decompose-C:** $(f \in \Sigma_{comm})$

$$\frac{\{E,\ f(t_1,t_2) = f(t'_1,t'_2)\}}{\{E,\ t_1 = t'_1, t_2 = t'_2)\} \vee \{E,\ t_1 = t'_2, t_2 = t'_1)\}}$$

- **Conflict:**

$$\frac{\{E,\ f(t_1,\ldots,t_n) = g(t'_1,\ldots,t'_m)\}}{\textbf{failure}}$$

if $f \neq g$

- **Coalesce:**

$$\frac{\{E,\ x = y\}}{\{E(x/y),\ x = y\}}$$

if $x, y \in \text{vars}(E),\ x \neq y$

Unification Modulo Commutativity (III)

- **Check:**

$$\frac{\{E,\ x = t\}}{\textbf{failure}}$$

if $x \in \mathsf{vars}(t),\ x \neq t$

- **Eliminate:**

$$\frac{\{E,\ x = t\}}{\{E(x/t),\ x = t\}}$$

if $x \notin \mathsf{vars}(t),\ t \notin V,\ x \in \mathsf{vars}(E)$.

Note that now, because of Rule **Decompose-C**, there can be several solutions to a unification problem. Also, we define **failure** as an identity element for $\_ \vee \_$.

We can illustrate the use of these rules by finding the most general unifiers modulo commutativity when $\Sigma_{comm} = \{g\}$.

Unification Modulo Commutativity (IV)

Let us apply these rules to solve the equation,

$$f(g(h(y),x),z) = f(z,g(k(u),v))$$

$\{f(g(h(y),x),z) = f(z,g(k(u),v))\} \longrightarrow$ (**Decompose**)

$\{g(h(y),x) = z,\ z = g(k(u),v)\} \longrightarrow$ (**Eliminate**)

$\{g(h(y),x) = g(k(u),v),\ z = g(k(u),v)\} \longrightarrow$ (**Decompose-C**)

$\{x = v,\ k(u) = h(y),\ z = g(k(u),v)\} \vee \{x = k(u),\ v = h(y),\ z = g(k(u),v)\} \longrightarrow$ (**Conflict** $\vee$ **Eliminate**)

failure $\vee$ $\{x = k(u),\ v = h(y),\ z = g(k(u),h(y))\} =$
$\{x = k(u),\ v = h(y),\ z = g(k(u),h(y))\}$

applying the resulting unifier we obtain the identity,

$f(g(h(y),k(u)),g(h(y),k(u))) =_{comm} f(g(k(u),h(y)),g(k(u),h(y)))$.

Where to Go from Here

We can only sketch how to go from here to more general unification algorithms, such as B-unification in an order-sorted signature Σ.

First of all, note that the presence of overloading and subsorts will typically move us from a single most general unifier to a finite set of them. This is because of the presence of subsort overloaded operators, which may lead to several different solutions. Note also that, in the presence of subsorts, even apparently innocent equations such as $x : s = y : s'$ may lead to **failure**, because the sorts s and s' may not have any common subsort. For example, in an `INT` specification, an equation `X = Y`, with `X` of sort `NzNat`, and `Y` of sort `NzNeg` will fail.

Where to Go from Here (II)

Chapter 15.1 of the Maude book gives a detail presentation of the inference rules for order-sorted C-unification and gives an implementation that you can use in Maude for experimentation.

Maude supports in a built-in way more general order-sorted unification algorithm allowing C, AC, and ACU axioms. This is documented in the Maude manual.

The Curch-Rosser Checker manual can be found in the Maude web page. It uses unification modulo any combinations of associativity, commutativity, and identity axioms, except associativity without commutativity; but can handle some cases of associativity without commutativity.

Where to Go from Here (III)

For a survey of unification algorithms modulo axioms see:

J.-P. Jouannaud and C. Kirchner, “Solving Equations in Abstract Algebras,” in J.-L. Lassez and G.Plotkin, eds., Computational Logic: Essays in Honor of Alan Robinson.

For order-sorted A-unification see:

J. Meseguer, J.A. Goguen, and G. Smolka, “Order-Sorted Unification,” *J. Symbolic Computation*, Volume 8, 1989, pages 383–413.

J. Hendrix and J. Meseguer, “Equational Order-Sorted Unification Revisited,” *Electr. Notes Theor. Comput. Sci.*, Vol. 290, 2012, pages 37–50.

What Are Critical Pairs?

The main result on citical pairs (generalizable to the modulo B case) is:

Theorem: Let (Σ, E) be an order-sorted equational theory, where the equations E are unconditional, with $\longrightarrow_E$ terminating. Then, E is confluent iff, for each pair of equations $u = v$ and $u' = v'$ in E (including equations $u = v$ considered twice) for each nonvariable position p in u, and for each most general order-sorted unifier θ such that $u_p\theta =_A u'\theta$, we have,

$$(\flat) \quad v\theta \downarrow_E u[v']_p\theta.$$

The corresponding equations $v\theta = u[v']_p\theta$, are called the critical pairs of the equations E,

What Are Critical Pairs? (II)

Proof: We had already reduced checking confluence to checking that, for each pair of equations $u = v$ and $u' = v'$ in E, for each nonvariable position p in u, and for each order-sorted unifier μ such that $u_p\mu = u'\mu$, we have,

$$(\flat) \quad v\mu \downarrow_E u[v']\mu.$$

But if $\{\theta_1, \ldots \theta_n\}$, are the most general order-sorted unifiers for the equation $u_p = u'$, then we can find a θ_i and a substitution ρ such that for all $x \in vars(u) \cup vars(u')$, $x\mu = x\theta_i\,\rho$.

We are then done if we prove the following:

What Are Critical Pairs? (III)

Substitution Lemma: If $t \overset{*}{\longrightarrow}_E t'$ and ρ is a substitution, then $t\rho \overset{*}{\longrightarrow}_E t'\rho$.

Proof: It is enough to prove the case for $t \longrightarrow_E t'$, since then the case $t \overset{*}{\longrightarrow}_E t'$ follows easily by induciton on the number of steps. But $t \longrightarrow_E t'$ means that there is an equation $u = v \in E$ and a substitution θ such that $t = t[u\theta]_p$ and $t' = t[v\theta]_p$. But then,

Note that, by the definition of the function $\_\rho$, we can easily prove that we have, $t\rho = t\rho[u\theta\rho]$, and $t'\rho = t'\rho[v\theta\rho]$. Therefore, $t\rho \longrightarrow_E t'\rho$, as desired.

q.e.d.

In Summary

What the Church-Rosser Checker does is:

- it checks that the equations E are sort-decreasing;
- it forms all the critical pairs for the equations E and tries to join them;
- it returns as proof obligations those equation specializations that it could not prove sort-decreasing, and those simplified critical pairs that it could not join.

The arguments in Lecture 6 and in this lecture have shown that this method is correct for checking confluence, under the termination assumption.

Checking Sufficient Completeness

We need methods to check that an equational theory (Σ, E) is sufficiently complete. For arbitrary equational theories sufficient completeness is in general undecidable. This is not so bad: it just means that we may have to do some inductive theorem proving.

Sufficient completeness is decidable for a very broad class of order-sorted theories, namely, unconditional theories of the form $(\Sigma, E \cup B)$ with B a set of axioms for operators allowing any combination of associativity and/or commutativity and/or identity, except associativity without commutativity, and E: (i) left-linear; (ii) ground confluent and sort-decreasing; and (iii) weakly terminating.

Checking Sufficient Completeness (II)

Furthermore, even for cases satisfying the above requirements (i)–(iii), but where B includes operators that are only associative, or associative and identity, sufficient completeness, although undecidable in theory, becomes decidable in practice for many specifications of interest using specialized heuristic algorithms.

Left-linearity (i) means that if $t = t' \in E$, then t has no repeated variables. This fails, e.g., for the idempotency equation $x \cup x = x$. Property (ii)–(iii) (modulo B) we are alredy familiar with.

Tree Automata

For equational theories satisfying the above requirements (i)–(iii) we can use decidability results from tree automata theory to cast the sufficient completeness problem into a tree automata problem and decide the problem that way.

An ordinary finite-state automaton $\mathcal{A}$ has a finite set Q of states and accepts strings of inputs when they lead the automaton to a subset $Q_0 \subseteq Q$ of accepting states. The language $L(\mathcal{A})$ of the automaton is then the set of all strings accepted by $\mathcal{A}$. Such languages are called regular languages and have nice decidablity results: they are closed under Boolean operations (we can construct automata for each such operation); and we can decide whether $L(\mathcal{A})$ is empty.

Tree Automata (II)

All this is generalized by finite-state tree automata, which accept terms in a term algebra $\mathcal{T}_\Sigma$ instead of just strings. A tree automaton is a tuple $\mathcal{A} = (\Sigma, Q, Q_0, R)$ with Σ an unsorted signature, Q a set of extra constants not in Σ called states, $Q_0 \subseteq Q$ a subset of accepting states, and R a set of transition rules, which can be of two forms:

- $f(q_1, \ldots, q_n) \longrightarrow q$, with $q_1, \ldots, q_n, q \in Q$, $f \in \Sigma$ and $f(q_1, \ldots, q_n) \in T_{\Sigma(Q)}$ (for $n = 0$ f can be a constant)
- $q \longrightarrow q'$, with $q, q' \in Q$ (epsilon transition)

Tree Automata (III)

Notice that we can view the transition rules R as ground rewrite rules and can use them to rewrite terms in the term algebra $\mathcal{T}_{\Sigma(Q)}$. Notice also that we have an inclusion $T_\Sigma \subseteq T_{\Sigma(Q)}$. We then define the language $L(\mathcal{A})$ as the subset $L(\mathcal{A}) \subseteq T_\Sigma$ of those $t \in T_\Sigma$ such that there is a $q \in Q_0$ such that $t \longrightarrow_R^* q$. A subset $L \subseteq T_\Sigma$ is called regular iff there is a finite-state tree automaton $\mathcal{A}$ such that $L = L(\mathcal{A})$.

Tree automata have the same decidablity results as string automata: they are closed under Boolean operations (we can construct automata for each such operation); and we can decide whether $L(\mathcal{A})$ is empty.

Tree Automata for Sufficient Completeness

The key observation is that, for theories $(\Sigma, E \cup B)$ satisfying conditions (i)–(iii), the following sets of ground Σ-terms are regular sets:

- the set D_s of terms of sort s having a defined symbol on top and constructor terms as arguments;
- the set C_s of constructor terms of sort s; and
- the set Red of terms reducible by the equations E (modulo B), i.e., terms not in normal form.

Under conditions (i)–(iii) $(\Sigma, E \cup B)$ is sufficiently complete iff for each sort s we have $D_s - (Red \cup C_s) = \emptyset$, which can be decided by deciding emptyness of the corresponding tree automaton.

Tree Automata Modulo B

Of course, in general we need to consider tree automata modulo B, that is, tuples $\mathcal{A} = (\Sigma, B, Q, Q_0, R)$, where (Σ, Q, Q_0, R) is an ordinary tree automaton, and B is a set of equational Σ-axioms such as associativity, commutativity, and identity of some symbols. The language of $\mathcal{A}$ is then a subset $L(\mathcal{A}) \subseteq T_{\Sigma/B}$, now defined by rewriting with R modulo B. That is, $L(\mathcal{A}) \subseteq T_{\Sigma/B}$ is the set of those $[t] \in T_{\Sigma/B}$ such that there is a $q \in Q_0$ such that $t \longrightarrow^{*}_{R/B} q$.

Hendrix, Ohsaki, and Viswanathan show that the needed tree automata decidablity results generalize to the modulo B case, for A any combination of associativity and/or commutativity and/or identity, except associativity alone, or associativity and identity alone.

Tree Automata Modulo A (II)

Even for the case of associativity alone, or associativity and identity alone, for which some tree automata questions like emptyness become undecidable, the sufficient completeness problem can still be decided in practice for many cases of interest by specialized heuristic algorithms (Hendrix, Ohsaki, and Viswanathan, Proc. RTA 2006, Springer LNCS).

All this means that in practice we can decide the sufficient completeness of most left-linear unconditional order-sorted specifications of interest.

An Example

To see how the desired tree automata needed to decide sufficient completeness can be built, we can use a simple example, our usual unsorted specification for addition for the Peano natural numbers with a single sort Nat, with 0 and s as constructors, and with equations $x + 0 = x$ and $x + s(y) = s(x + y)$. This specification satisfies conditions (i)–(iii), since it is left-linear, confluent, sort-decreasing, and terminating.

To recognize each of the regular sets Red, D_{Nat}, and C_{Nat} we need three tree automata $\mathcal{A}_{Red}$, $\mathcal{A}_{D_{Nat}}$, and $\mathcal{A}_{C_{Nat}}$.

An Example (II)

The tree automata $\mathcal{A}_{Red}$, $\mathcal{A}_{D_{Nat}}$, and $\mathcal{A}_{C_{Nat}}$ have all the same signature Σ (namely that of the natural numbers), set of states $Q = \{Nat, Red, D_{Nat}, C_{Nat}, q_0, q_{s(y)}\}$, and transitions R:

- $s(Nat) \longrightarrow Nat$, $Nat + Nat \longrightarrow Nat$, and epsilon transitions $C_{Nat}, Red, D_{Nat}, q_0, q_{s(y)} \longrightarrow Nat$
- (constructor transitions): $0 \longrightarrow C_{Nat}$, $s(C_{Nat}) \longrightarrow C_{Nat}$
- (defined function transition): $C_{Nat} + C_{Nat} \longrightarrow D_{Nat}$
- (reducibility transitions): $0 \longrightarrow q_0$, $s(Nat) \longrightarrow q_{s(y)}$, $Nat + q_0 \longrightarrow Red$, $Nat + q_{s(y)} \longrightarrow Red$.

They only differ in their respective accepting state: Red, D_{Nat}, and C_{Nat}.

An Example (III)

The point now is that each Boolean operation on regular tree languages has a corresponding operation on their associated tree automata. Therefore, out of the automata $\mathcal{A}_{Red}$, $\mathcal{A}_{D_{Nat}}$, and $\mathcal{A}_{C_{Nat}}$ we can construct an automaton that recognizes the language $D_{Nat} - (Red \cup C_{Nat})$. Let us call this automaton $\mathcal{A}_{D_{Nat}-(Red \cup C_{Nat})}$. Now we know that under conditions (i)–(iii) our specification is sufficiently complete iff $D_{Nat} - (Red \cup C_{Nat}) = \emptyset$. Therefore, we can decide this property by testing $\mathcal{A}_{D_{Nat}-(Red \cup C_{Nat})}$ for emptyness. If the test (as for this example) succeeds, we are done. If it doesn't, we get very useful counterexample terms, showing us where sufficient completeness fails.

The Maude SCC Tool

The Maude Sufficient Completeness Checker (SCC) is a tool developed by Joseph Hendrix at UIUC. It uses a library of tree automata modulo B operations also developed by him, and reduces the sufficient completeness problem of specification $(\Sigma, E \cup B)$ satisfying conditions (i)–(iii) to the emptyness problem for the tree automaton $\mathcal{A}_{D_s-(Red \cup C_s)}$ for each sort s in Σ. It outputs either "success" or a set of counterexample terms.

Instructions to acces SCC can be found in the course web page. Its use is essentially very simple. One: (1) loads the module `scc.maude`; (2) loads the module to be checked, say `FOO`; (3) types "`select SCC-LOOP .`" and "`loop init-scc .`" and (4) gives to the SCC the command "`(scc FOO .)`".

The Maude SCC Tool (II)

We can illustrate the use of the SCC with some examples already encountered previously in the course. Consider the module

```
fmod NATURAL is
sort Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op _+_ : Nat Nat -> Nat .
vars X Y : Nat .
eq X + 0 = X .
eq X + s(Y) = s(X + Y) .
endfm
```

The Maude SCC Tool (III)

This module is indeed successfully checked by SCC:

```
Maude> load scc .
Maude> in natural .
==========================================
fmod NATURAL
Maude> select SCC-LOOP .
Maude> loop init-scc .
Starting the Maude Sufficient Completeness Checker.
Maude> (scc NATURAL .)
Checking sufficient completeness of NATURAL ...
Warning: This module has equations that are not
    left-linear. The sufficient completeness checker will
    rename variables as needed to drop the non-linearity
    conditions.
Success: NATURAL is sufficiently complete under the
    assumption that it is weakly-normalizing, confluent,
    and sort-decreasing.
```

The Maude SCC Tool (IV)

Consider the module

```
fmod MY-LIST is
  protecting NAT .
  sorts NzList List .
  subsorts Nat < NzList < List .
  op _;_ : List List -> List [assoc] .
  op _;_ : NzList NzList -> NzList [assoc ctor] .
  op nil : -> List [ctor] .
  op rev : List -> List .
  eq rev(nil) = nil .
  eq rev(N:Nat) = N:Nat .
  eq rev(N:Nat ; L:List) = rev(L:List) ; N:Nat .
endfm
```

The Maude SCC Tool (V)

when checked by the SCC gives us the counterexample

```
Maude> load scc
Maude> in mylist
==========================================
fmod MY-LIST
Maude> select SCC-LOOP .
Maude> loop init-scc .
Starting the Maude Sufficient Completeness Checker.
Maude> (scc MY-LIST .)
Checking sufficient completeness of MY-LIST ...
Warning: This module has equations that are not
    left-linear. The sufficient completeness checker will
    rename variables as needed to drop the non-linearity
    conditions.
Failure: The term 0 ; nil is a counterexample as it is a
    irreducible term with sort List in MY-LIST that does
    not have sort List in the constructor subsignature.
```

The Maude SCC Tool (VI)

We can correct this problem revising our module:

```
fmod MY-LIST2 is
  protecting NAT .
  sorts NzList List .
  subsorts Nat < NzList < List .
  op _;_ : List List -> List [assoc] .
  op _;_ : NzList NzList -> NzList [assoc ctor] .
  op nil : -> List [ctor] .
  op rev : List -> List .
  eq rev(nil) = nil .
  eq rev(N:Nat) = N:Nat .
  eq rev(N:Nat ; L:List) = rev(L:List) ; N:Nat .
  eq nil ; L:List = L:List .
  eq L:List ; nil = L:List .
endfm
```

The Maude SCC Tool (VII)

which is now successfully checked by SCC:

```
Maude> load scc
Maude> in mylist2
==========================================
fmod MY-LIST2
Maude> select SCC-LOOP .
Maude> loop init-scc .
Starting the Maude Sufficient Completeness Checker.
Maude> (scc MY-LIST2 .)
Checking sufficient completeness of MY-LIST2 ...
Warning: This module has equations that are not
    left-linear. The sufficient completeness checker will
    rename variables as needed to drop the non-linearity
    conditions.
Success: MY-LIST2 is sufficiently complete under the
    assumption that it is weakly-normalizing, confluent,
    and sort-decreasing.
```