Program Verification: Lecture 9

José Meseguer

Computer Science Department
University of Illinois at Urbana-Champaign

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 $\vec{E}$: (i) left-linear; (ii) sort-decreasing; and (iii) 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, i.e., if $x \in vars(t)$, then x occurs at a single position p in t. This fails, e.g., for the idempotency equation $x \cup x = x$, where x occurs at positions 1 and 2 in $x \cup x$. Properties (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 an unsorted term algebra $\mathbb{T}_\Sigma$ instead of just strings. A tree automaton (TA) 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:

- $f(q_1, \ldots, q_n) \to q$, 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 will be a constant, and the rule becomes $f \to q$)

- $q \to q'$, 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 $\mathbb{T}_{\Sigma(Q)}$. I.e., the TA specifies a ground TRS $(\Sigma(Q), R)$ (plus the subset $Q_0 \subseteq Q$). 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 $t \to_R^* q$. 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 as Order-Sorted Signatures

Automata are labeled graphs. Tree automata are labeled multigraphs, actually, order-sorted signatures. Any tree automaton $\mathcal{A} = (\Sigma, Q, Q_0, R)$, with $\Sigma = (\{s\}, F, \Sigma)$, is the order-sorted signature: $\Sigma_{\mathcal{A}} = ((Q, <), F, \Sigma_{\mathcal{A}})$, where,

$$q < q' \;\Leftrightarrow\; q \to^{+}_{R_\epsilon} q'$$

where R_ϵ are the epsilon transitions (w.l.o.g. may assume R_ϵ terminating). And for each $q_1, \ldots, q_n, q \in Q$ we have:

$$f(q_1, \ldots, q_n) \to q \;\; in\; R \quad \Leftrightarrow \quad f : q_1, \ldots, q_n \to q \;\; in\; \Sigma_{\mathcal{A}}$$

If the accepting states are $Q_0 = \{q_1, \ldots, q_k\}$, then the regular language $L(\mathcal{A})$ is:

$$L(\mathcal{A}) = T_{\Sigma_{\mathcal{A}}, q_1} \cup \ldots \cup T_{\Sigma_{\mathcal{A}}, q_k}.$$

This is because, for each $q \in Q$, $t \in T_{\Sigma_{\mathcal{A}}, q}$ iff $t \to^{*}_{R} q$

Tree Automata for Sufficient Completeness

For checking 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 emptiness of the corresponding tree automaton.

Tree Automata Modulo B

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

Hendrix, Ohsaki, and Viswanathan extend the tree automata decidablity results to the modulo B case, for B any combination of associativity and/or commutativity and/or identity, except associativity without commutativity.

Tree Automata Modulo B with A and not C Axioms

Even in the case when B contains axioms for a binary operator f that is associative (A) or AU, but not commutative, for which some tree automata questions like emptiness 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 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 $\mathcal{A}_{Ctor}$ have the same signature Σ (that of the natural numbers), set of states $Q = \{Nat, Red, D_{Nat}, Ctor, Zero, NzCtor\}$, and transitions R:

- (ϵ-transitions): $Ctor \to Nat,\ Red \to D_{Nat},\ D_{Nat} \to Nat,$ $Zero \to Ctor,\ NzCtor \to Ctor.$
- (Nat transitions) $s(Nat) \to Nat$, $Nat + Nat \to Nat.$
- (constructor transitions): $0 \to Zero$, $s(Ctor) \to NzCtor.$
- (defined function transition): $Ctor + Ctor \to D_{Nat}.$
- (reducibility transitions): $Ctor + Zero \to Red,$ $Ctor + NzCtor \to Red.$

An Example (III)

$\mathcal{A}_{Red}$, $\mathcal{A}_{D_{Nat}}$ and $\mathcal{A}_{Ctor}$ only differ in their respective accepting state: Red, D_{Nat}, and $Ctor$. Since their: (i) signature, (ii) set of states and (iii) transitions are the same, the shared part (i)–(iii) is just the order sorted signature:

```
sorts Nat Red, D-Nat Ctor Zero NzCtor .
subsorts Zero NzCtor < Ctor < Nat .
subsorts Red < D-Nat < Nat .
op s : Nat Nat -> Nat .
op s : Ctor -> NzCtor .
op 0 : -> Zero .
op _+_ : Nat Nat -> Nat .
op _+_ : Ctor Ctor -> D-Nat .
op _+_ : Ctor Zero -> Red .
op _+_ : Ctor NzCtor -> Red .
```

An Example (IV)

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 $\mathcal{A}_{Ctor}$ we can construct an automaton that recognizes the language $D_{Nat} - (Red \cup Ctor)$. Let us call this automaton $\mathcal{A}_{D_{Nat}-(Red \cup Ctor)}$. Now we know that under conditions (i)–(iii) our specification is sufficiently complete iff $D_{Nat} - (Red \cup Ctor) = \emptyset$. Therefore, we can decide this property by testing $\mathcal{A}_{D_{Nat}-(Red \cup Ctor)}$ for emptiness. 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 emptiness 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. The module `NATURAL` was already used above as an example to illustrate the tree automata $\mathcal{A}_{Red}$, $\mathcal{A}_{D_{Nat}}$ and $\mathcal{A}_{Ctor}$. The SCC tool will now check the emptiness of $\mathcal{A}_{D_{Nat}-(Red \cup Ctor)}$.

```
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)

Emptiness of $\mathcal{A}_{D_{Nat}-(Red \cup Ctor)}$. is 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.
```