Formal Methods: 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 proving sufficient completeness, by being a property of the initial algebra $T_{\Sigma/E}$, may in general require doing some inductive theorem proving.

Sufficient completeness is decidable for a very broad class of order-sorted theories, namely, unconditional theories of the form $(\Sigma, E \cup A)$ with A a set of axioms for operators allowing any combination of associativity and/or commutativity and/or identity, except associativity alone, or associativity and identity alone, 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 A 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) (modulo A) we are alredy familiar with. Property (iii), weak termination, means that we only require that each term t has a canonical from $can(t)$ such that $t \longrightarrow^*_{E/A} can(t)$. For example, E with $f(x) = f(f(x))$ and $f(x) = a$ is nonterminating but is weakly terminating.

Tree Automata

For equational theories 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 T_Σ 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 trewrite terms in the term algebra $T_{\Sigma(Q)}$. 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 $(\Sigma, E \cup A)$ 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 A), i.e., terms not in canonical form.

Under conditions (i)–(iii) $(\Sigma, E \cup A)$ 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 A

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

Hendrix, Ohsaki, and Viswanathan show that the needed tree automata decidablity results generalize to the modulo A 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).

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 $Q = \{Red, D_{Nat}, C_{Nat}, q_0, q_{s(y)}\}$, and transitions R:

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

The 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 A operations also developed by him, and reduces the sufficient completeness problem of specification $(\Sigma, E \cup A)$ satisfying conditions (i)–(iii) to the emptyness problem for the tree automata $\mathcal{A}_{D_{s-(Red \cup C_s)}}$ 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.
```

Verification of Maude Functional Modules

We are now ready to begin discussing program verification for deterministic declarative programs, and, more specifically, for functional modules in Maude.

Such functional modules are of the form $\texttt{fmod}(\Sigma, E \cup A)\texttt{endfm}$, where we assume E confluent, sort-decreasing, and terminating modulo A. Their mathematical semantics is given by the initial algebra $T_{\Sigma/E \cup A}$.

Their operational semantics is given by equational simplification with E modulo A. Both semantics coincide in the canonical term algebra, since we have the Σ-isomorphism,

$$T_{\Sigma/E \cup A} \cong Can_{\Sigma, E/A}.$$

Verification of Maude Functional Modules (II)

What are properties of a module $\texttt{fmod}(\Sigma, E \cup A)\texttt{endfm}$?

They are sentences φ, perhaps in equational logic, or, more generally, in first-order logic, in the language of a signature containing Σ.

When do we say that the above module satisfies property φ?

When we have,

$$T_{\Sigma/E \cup A} \models \varphi.$$

How do we verify such properties?

A Simple Example: Associativity of Addition

Consider the module,

```
fmod NATURAL is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M L : Natural .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

A property φ satisfied by this module is the associativity of addition, that is, the equation,

$$(\forall \mathtt{N}, \mathtt{M}, \mathtt{L})\ \mathtt{N} + (\mathtt{M} + \mathtt{L}) = (\mathtt{N} + \mathtt{M}) + \mathtt{L}.$$

Need More than Equational Deduction

Since the initial model $T_{\Sigma/E \cup A}$ associated to a module $\texttt{fmod}(\Sigma, E \cup A)\texttt{endfm}$ satisfies the equations $E \cup A$, by the **Soundness Theorem** for equational deduction, whenever we can prove an equation φ by $E \cup A \vdash \varphi$, we must have $T_{\Sigma/E \cup A} \models \varphi$, and therefore the module satisfies φ.

Therefore, equational deduction is always a sound proof method to verify properties of functional modules. However, it is quite limited, and generally insufficient for many properties.

In particular, for φ the associativity of addition and E the equations in `NATURAL` (in this case $A = \emptyset$) we cannot prove $E \vdash \varphi$.

Need More than Equational Deduction (II)

This is easy to see, since associativity is not a property satisfied by all models of E. Consider, for example, the initial model obtained by adding a nonstandard number `a`,

```
fmod NON-STANDARD-NAT is
  sort Natural .
  ops 0 a : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M L : Natural .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

Since it has the same equations E, this initial model satisfies E, but it does not satisfy associativity, since `a +(a + a)` $\neq$ `(a + a)+ a`. In fact, no equations apply to either side.

Inductive Properties

The point is that associativity is an inductive property of natural number addition; that is, one satisfied by the initial model of E, but not in general by other models of E.

What we need are inductive proof methods based on a more powerful proof system $\vdash_{ind}$, satisfying the soundness requirement,

$$E \cup A \vdash_{ind} \phi \quad \Rightarrow \quad T_{\Sigma/E \cup A} \models \phi.$$

Also, it should prove all that equational deduction can prove and more. That is, for formulas φ that are equations it should satisfy,

$$E \cup A \vdash \phi \quad \Rightarrow \quad E \cup A \vdash_{ind} \phi.$$

Inductive Properties (II)

Because of Gödel's Incompleteness Theorem, in general we cannot hope to have completeness of inductive inference, that is, to have an equivalence

$$E \cup A \vdash_{ind} \phi \quad \Leftrightarrow \quad T_{\Sigma/E \cup A} \models \phi.$$

The inductive inference system that we will justify and use generalizes the usual proofs by natural number induction. In fact, in our example of associativity of natural number addition it actually specializes to the usual proof method by natural number induction.