Reasoning by Rewriting

CafeOBJ Team of JAIST

Topics

- Introduction to the theory of term rewriting systems, which is a basis of the CafeOBJ execution
- How to write CafeOBJ specifications which satisfy two of the most important properties of TRS:
  - Termination
  - Confluence

Overview

- In the first half, we treat simple equational specifications which consist of
  - ordinary operators without any attribute and
  - equations without conditions
- In the last half, we discuss on rewriting for specifications including
  - operators with associative and commutative attributes and
  - conditional equations

Term rewriting system

Term rewriting system

- **The term rewriting system** (**TRS**) gives us an efficient way to prove equations by regarding an equation as a left-to-right rewrite rule
- **Rewriting** is the replacement of a **redex** with the corresponding instance of the rhs
  - A redex is an instance of the lhs of an equation
  - e.g.

```
s (0 + s 0) --> s (s (0 + 0))
```

```
NAT+
eq N + 0 = N .
eq M + s N = s (M + N) .
```


Equational reasoning by TRS

- A reduction is a process of rewriting from a given term to a normal form
  - A normal form is a term which cannot be rewritten
- Equational reasoning by TRS is done by reducing both sides of a given equation and comparing their normal forms

```
0 + s 0 --> s (0 + 0) --> --> s 0
s 0 + 0 --> s 0
```

```
NAT+
eq N + 0 = N .
eq M + s N = s (M + N) .
```

Equational reasoning with EQL

- A built-in module EQL is useful to check joinability of given terms

  A special predicate _=_ is defined for all sorts

```
NAT+ + EQL> red 0 + s 0 = s 0 + 0 .
[1]: ((0 + (s 0)) = ((s 0) + 0))
[2]: ((s (0 + 0)) = ((s 0) + 0))
[3]: ((s 0) = ((s 0) + 0))
[4]: ((s 0) = (s 0))
---> true
(true):Bool
```

SOUNDNESS:
If s = t is reduced into true, it holds in all models

Conditions of TRS

- Rewrite rules should satisfy the following conditions on variables
  - Any lhs should not be a variable
    - Such a rule, e.g. **N = N + 0**, causes an infinite loop

```
s 0 --> s 0 + 0 --> (s 0 + 0) + 0 --> ...
```

  - Any variable in rhs should appear in lhs
    - By such a rule, e.g. **0 = N * 0**, a redex can be rewritten into infinitely many terms

```
0 --> 0 * 0
0 --> s 0 * 0 ...
```

Bad equations ignored

- CafeOBJ system uses only equations satisfying the variable conditions when reducing terms by the reduction command

Properties of TRS

- TRS achieves only a partial equational reasoning, in general, because equations are directed
  - e.g. `b = c` cannot be proved by TRS {`a = b`, `a = c`}
- However, TRS can prove any equation which can be deduced from the axiom E of SP if SP has the termination and confluence properties

Termination

Definition of Termination

- A specification (a TRS or a set of equations) SP is **terminating** if and only if there is no infinite rewrite sequence `t0 --> t1 --> t2 --> ...`
- Termination guarantees that any term has a normal form, and makes us possible to compute a normal form in finite times

NAT-COM

```
eq X + Y = Y + X .
```

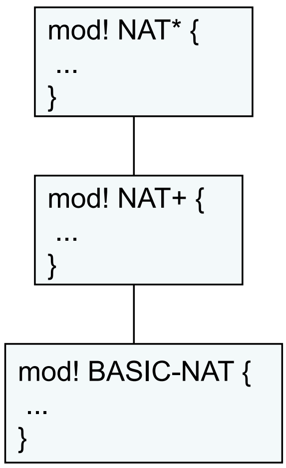
```
s 0 + 0 --> 0 + s 0 --> s 0 + 0 --> ...
```

Proving termination

- Termination is an undecidable property, i.e. no algorithm can decide termination of term rewriting systems
- Several sufficient conditions for termination have been proposed
- In this presentation, we give one way to write terminating specifications

Hierarchical design

- A hierarchical design of a specification of an abstract data type SP consists of
  - Module **BASIC-SP** for functions' domain and range
  - Module **SP-F$_0$** importing BASIC-SP for defining a function F_0
  - Module **SP-F$_{i+1}$** importing SP-F$_i$ for defining a function F_{i+1} which is defined using a function F_k ($k < i + 1$)



BASIC-SP

- An operator in BASIC-SP is called a constructor
- Constructor terms denote elements of the domain
  - A constructor term is a term consisting of only constructors

```
mod! BASIC-NAT {
 [Zero NzNat < Nat]
 op 0  : -> Zero
 op s_ : Nat -> NzNat
}
```

```
0,  s 0,  s s 0, s s s 0, ...
```


SP-F_0

- SP-F_0 consists of a protecting import of BASIC-SP, an operator F_0, and equations defining F_0
- Each rhs should be constructed from **variables**, constructors, and recursive calls
  - F(.,t,.) is a recursive call of F(.,t',.) iff t is a subterm of t'

```
mod! NAT+ {
 pr(BASIC-NAT)
 op _+_  : Nat Nat -> Nat
 vars M N : Nat
 eq N + 0 = N .
 eq M + s N = s (M + N) .
}
```

SP-F_{i+1}

- SP-F_{i+1} consists of a protecting import of SP-F_i, an operator F_{i+1}, and equations defining F_{i+1}
- Each rhs should be constructed from **variables**, constructors, pre-defined functions F_k ($k < i + 1$), and recursive calls

```
mod! NAT* {
 pr(NAT+)
 op _*_  : Nat Nat -> Nat
 vars M N : Nat
 eq N * 0 = 0 .
 eq M * s N = M + (M * N).
}
```

```
mod! NAT-FACT {
 pr(NAT*)
 op fact_  : Nat -> Nat
 vars M N : Nat
 eq fact 0 = s 0 .
 eq fact (s N) = s N * (fact N) .
}
```

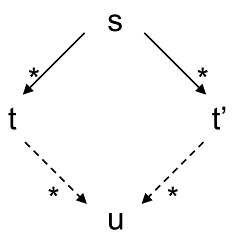

Recursive Path Order

- RPO is one of the most famous classic termination proof techniques
  - By RPO, we can prove termination of specifications described according to the hierarchical design
- For a specification beyond the hierarchical design, you may find useful termination provers on Internet: AProVE, CiME, TTT, etc

Confluence

Definition of Confluence

- SP is **confluent** iff all divided terms are joinable, i.e., if `s -->* t` and `s -->* t'` then `t -->* u` and `t' -->* u` for some `u`
  - `-->*` denotes zero or many rewrite steps



NAT-ASSOC

```
eq (X + Y) + Z = X +(Y + Z) .
eq first(X + Y) = X .
```

```
first((0 + s 0) + s s 0) --> 0 + s 0

first((0 + s 0) + s s 0)
--> first(0 + (s 0 + s s 0)) --> 0
```

Termination and Confluence

- Confluence guarantees that a normal form is unique for any term
- Thus, for a terminating and confluent SP, any term has the unique normal form
- We obtain complete equational reasoning:
  - Reduce both sides of a given equation
  - Compare their normal forms
    - The equation is deducible from the axiom if they are same
    - It is not deducible if they are not

Branch

- It is trivial that SP without any branch is confluence
- Unfortunately, such a SP is rare because an operator with more than one arities can include more than one redexs
  - (Assume `a --> b`) **`f(b, a) <-- f(a, a) --> f(a, b)`**
- Fortunately, such branches can be recovered by rewriting redexs of each other rewrite
  - **`f(b, a) --> f(b, b) <-- f(a, b)`**
- What branches are troublesome?

Overlap

- Terms overlap iff a one's instance is an instance of the other's non-variable subterm
  - (X + Y) + Z is an instance of X + Y of first(X + Y)
  - A branch resulting from an overlap may not be recovered because a redex may disappear

NAT-ASSOC

```
eq (X + Y) + Z = X +(Y + Z) .
eq first(X + Y) = X .
```

```
first((0 + s 0) + s s 0) --> 0 + s 0
first((0 + s 0) + s s 0)
--> first(0 + (s 0 + s s 0)) --> 0
```


Overlapping rewrite rules

- Rewrite rules overlap if their lhss overlap
- SP overlaps if there are overlapping rewrite rules
  - You can take two copies of one rewrite rule to check an overlap. For such cases, the overlap at the root position should be ignored
  - e.g. a rewrite rule **`~ ~ x = x`** overlaps itself because **`~ ~ x`** is an instance of a subterm **`~ x`**
- A unifier of two overlapping terms (`s`, `t`) is an instance of `s` which has a `t`'s instance
  - e.g. **`~ ~ ~ 0`** is a unifier of (**`~ ~ x`**, **`~ ~ x`**)

Critical Pair

- The most general unifier of overlapping rewrite rules has two direct descendant. Such a pair is called a critical pair

BOOL-NOT

```
eq ~ ~ X = X .
eq   ~ 0 = 1 .
eq   ~ 1 = 0 .
```

  - The m.g.u. of **`~ ~ X`** and **`~ 0`** is **`~ ~ 0`**
  - The CP of them is (**`0, ~1`**) because **`~ ~ 0 --> 0`** by the 1st rule and **`~ ~ 0 --> ~ 1`** by the 2nd rule

Sufficient condition of Confluence

- Theorem (Knuth and Bendix 1970): If SP is terminating and all critical pairs are joinable, then SP is confluent

BOOL-NOT

```
eq ~ ~ X = X .
eq   ~ 0 = 1 .
eq   ~ 1 = 0 .
```

  - **`BOOL-NOT`** has three CPs: **(0, ~1)**, **(1, ~0)** and **(~ X, ~X)**, and all those CPs are joinable, thus, it is confluent

Conditional Equations

Conditional equations

- CafeOBJ allows us to write a condition for an equation
  - A condition is a term of Boolean sort `Bool`
  - CafeOBJ modules import a built-in Boolean module `BOOL` implicitly, thus, you can use Boolean operators to write equations

NAT-EVEN

```
eq even 0 = true .
ceq even(s N) = false if even N .
ceq even(s N) = true  if not (even N) .
```

Reduction by conditional equations

- A conditional equation is applied when the condition part is reduced into true

```
NAT-EVEN> red even s 0 .
-- reduce in NAT-EVEN : (even (s 0)):Bool
1>[1] apply trial #1
-- rule: ceq (even (s N:Nat)) = false if (even N)
    { N:Nat |-> 0 }
2>[1] rule: eq (even 0) = true
    {}
2<[1] (even 0) --> true
1>[2] match success #1
1<[2] (even (s 0)) --> false
(false):Bool
```

Try to apply the cond. equation

The condition part is reduced into true

Apply the equation part

Termination of conditional equations

- To obtain a terminating conditional SP, not only rhs but a condition part should also be cared

NAT-EVEN

```
ceq even(s N) = false if even N .
ceq even(s N) = true  if not (even N) .
```

INFINITE

```
ceq f(X) = true  if f(X) .
```

```
INFINITE> red f(X:Elt) .
-- reduce in INFINITE : (f(X)):Bool
[Warning]:
Infinite loop? Evaluation of condition nests too deep,
terminates rewriting: f(X:Elt)
INFINITE>
```

Confluence of conditional equations

- In most cases, conditional SPs overlap because conditions are used to write case-splitting of a same pattern

NAT-EVEN

```
ceq even(s N) = false if even N .
ceq even(s N) = true  if not (even N) .
```

  - For confluence, each condition of a pattern should be separated from each other, i.e., if one is true, then the others should be false, for example,
    - P(X), not P(X)
    - X < 5, ((5 <= X and X < 10), 10 <= X

Associative Commutative Operators

Associative Commutative operators

- Equations of Associativity and Commutativity may cause non-termination and non-confluence
- They are recommended to be specified as operators attributes

```
eq X + Y = Y + X .
eq (X + Y) + Z = X + (Y + Z) .
```

NAT+AC

```
op _+_ : Nat Nat -> Nat { assoc comm }
```

- You do not need bracket for associative operators
- eq N + 0 = N can be applied to (N + 0) since + is commutative.

```
NAT+AC> red 0 + (N:Nat) + 0 .
N:Nat
```


Specification of bags (multi-sets)

BAG

```
[Elt < Bag]
ops a b c  : -> Elt
op _ _ : Bag Bag -> Bag { assoc comm }
op _in_ : Elt Bag -> Bool
var E : Elt    var B : Bag
eq E in (E B) = true .
```

- From the subsort relation [**`Elt < Bag`**] and the associative operator (_ _), a sequence of **`Elt`** is a term of **`Bag`**

```
c in (a b c)   =   c in (a (c b))
               =   c in ((c b) a)
               =   c in (c (b a)
               =   true
```

AC Rewriting

- One step AC (or A or C) Rewriting, denoted by -->_{AC}, is defined as the composition ($=_{AC}$ o -->)

```
c in (a b c)    =C   c in (a (c b))
                =C   c in ((c b) a)
                =A   c in (c (b a))
                -->  true
```

When applying a rewrite rule to a term with AC operators, first compute all AC equivalent terms (it is finite), and if there is a redex, then rewrite it

a (b c), (a b) c, a (c b), (a c) b, b (a c), (b a) c, b (c a), (b c) a, **c (a b)**, (c a) b, **c (b a)**, (c b) a

Termination of AC Rewriting

- Even if SP is terminating, adding AC attribute to some operator makes it non-terminating

BAG2

```
  [Elt < Bag]
ops 0 1  : -> Elt
op _ _  : Bag Bag -> Bag { assoc comm }
var E : Elt
eq (E E) = 0 1 .
```

```
0 (0 1)   =A    (0 0) 1
          -->   (0 1) 1
           =A   0 (1 1)
          -->   0 (0 1) ...
```

Confluence of AC Rewriting

- Even if SP is confluent, adding AC attribute to some operator makes it non-confluent

BAG3

```
ops 0 1  : -> Elt
op begin-with-0 : Bag -> Bool
op _ _ : Bag Bag -> Bag { assoc comm }
var B : Bag
eq begin-with-0(0 B) = true .
eq begin-with-0(1 B) = false .
```

```
begin-with-0(0 1) --> true

begin-with-0(0 1) =C begin-with-0(1 0) --> false
```


Summary

- For a given equation, [Reducible by rewriting] => [Deducible from E] => [Satisfied by any model], however,
  - The opposite is not true in general
  - Reducible <=> Deducible holds when it is terminating and confluent
- To obtain a terminating SP, describe it according to the hierarchical design with recursive definition
- To obtain a confluence SP, check all critical pairs are joinable

References

- F. Baader and T.Nipkow, **Term Rewriting and all that**, Cambridge Univ. Press, 1998.
  - Introduction to TRS: Termination, Confluence
- E.Ohlebusch, **Advanced topics in Term Rewriting**, Springer, 2002.
  - + Conditional TRS, Modularity
- Terese, **Term Rewriting systems**, Cambridge Univ. Press, 2003.
  - + Strategy, Higher-order rewriting
- **AProVE** : http://aprove.informatik.rwth-aachen.de/
  - System for automated termination, supports Conditional TRS, AC-TRS, etc

Extra topic

- Sufficient completeness

Sufficient completeness

- A function f is **sufficiently complete** if and only if for any constructors arguments $t_1,\ldots,t_n$, the term $f(t_1,\ldots,t_n)$ is equivalent to some constructor term t
  - That is, $f(t_1,\ldots,t_n) = t$ can be deduced from the axiom

NAT+
```
eq N + 0 = N .
eq M + s N = s (M + N) .
```

NAT+x
```
eq 0 + N = N .
eq M + s N = s (M + N) .
```

(s 0) + 0 is un-defined

Sufficient condition of sufficient completeness

- SP is sufficiently complete if
  - SP is terminating, and
  - All function operators are reducible, that is, for any ground (variable-free) term which includes a function operator, it is reducible (= a redex exists)
  - E.g. s (s (0 + (0 + s 0)))

NAT+
```
eq N + 0 = N .
eq M + s N = s (M + N) .
```

  - Because of the 1st condition each term has its normal form, and
  - Because of the 2nd condition each normal form is constructed by constructors only

Into one module

- If all functions are defined sufficiently complete, they can be written into one module without changing its denotation
  - Actually, specifications of data types are often described in one module including constructors and functions together

```
mod! NAT-fact{
  [Zero NzNat < Nat]
  op 0 : -> Zero {constr}
  op s_ : Nat -> NzNat {constr}
  op _+_ : Nat Nat -> Nat
  op _*_ : Nat Nat -> Nat
  op fact_ : Nat -> Nat
```

```
vars M N : Nat
  eq N + 0 = N .
  eq M + s N = s(M + N) .
  eq N * 0 = 0 .
  eq M * s N = (M * N) + M .
  eq fact 0 = s 0 .
  eq fact (s N) = (s N) * (fact N) .
}
```

Sinaia School Lec