Reasoning by Rewriting

NAKAMURA Masaki
Kanazawa University

Topics

- Introduction to the theory of term rewriting systems (TRS), which gives a basis of executable algebraic specification languages
- How to write executable specifications well based on the theory of TRS

Overview

- Introduce the fundamental properties of TRS with examples of simple specifications
  (no operators attributes, no conditional equations)
  - Equational reasoning by TRS
  - Termination and Confluence
- Discuss on more practical specifications, which includes
  - Conditional equations
  - Operators attributes (associative and commutative)

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

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

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

```
s 0 + s s 0          =?=          s s 0 + s 0
s (s 0 + s 0)                     s (s s 0 + 0)
s s (s 0 + 0)
                  s s s 0
```

Equational reasoning with EQL

- A built-in module EQL can be used for checking joinability of given terms
  A special predicate _=_ is defined for all sorts in EQL

```
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
```

  - The equational reasoning with _=_ is sound, that is, If s = t is reduced into true, it holds in all models

Reduction

- A redex is an instance of the lhs of an equation
  - (1-step) Rewriting is a replacement of a redex with the corresponding instance of the rhs
  - e.g.

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

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

- A normal form is a term which cannot be rewritten
  - A reduction is a sequence of rewriting from a given term to a normal form

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

Variable conditions for 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

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

Termination

Definition of Termination

- A specification (a TRS or a set of equations) SP is terminating if there is no infinite rewrite sequence t_0 --> t_1 --> t_2 --> ...
- Termination guarantees that any term has a normal form, and makes us possible to compute a normal form in finite times

```
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 whether a given term rewriting system is terminating
- Several sufficient conditions for termination have been proposed
  - Recursive path order (RPO) is one of the most well-known classical termination methods.
  - We give a way to write specifications whose termination can be proved by RPO

Hierarchical design

- A hierarchical design of a specification of an abstract data type *SP* consists of
  - Module BASIC-*SP*
    - for functions' domain and/or 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} by using functions F_k ($k < i + 1$)

```
mod! NAT* {
 pr(NAT+)
 op _*_ : ...
}
```

```
mod! NAT+ {
 pr(BASIC-NAT)
 op _+_ : ...
}
```

```
mod! BASIC-NAT {
 ...
}
```

BASIC-SP

- An operator in BASIC-SP is called a constructor
  - A constructor term is a term consisting of only constructors
- Constructor terms denote elements of the carrier set (denoted by a sort)

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

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

Recursive call

- If every function is recursively defined, the SP is terminating
- Recursive calls are defined as follows:
  - $F_i(t_1,\ldots t_n)$ is a recursive call of $F_i(l_1,\ldots l_n)$ if
    - All arguments $\boldsymbol{t_i}$ are subterms of $\boldsymbol{l_i}$ resp., and
    - One of the arguments $\boldsymbol{t_i}$ is a strict subterm of $\boldsymbol{l_i}$.

```
M + s N

M +   N
```

```
equal(s M, s N)

eqaul(M,   N)
```

```
  M + s N

s M + N
```

SP-F_i

- SP-F_i consists of a protecting import of BASIC-SP ($i = 0$) or SP-F_j ($i > j >= 0$), an operator F_i and equations for F_i
  - Each lhs has F_i at the root position
  - Each rhs is constructed from variables, constructors, pre-defined functions and recursive calls (of the lhs) only

```
mod! NAT+ {
 pr(BASIC-NAT)
  ...
 eq N +   0 = N  .
 eq M + s N = s (M + N)  . }
```

$F_0 =$ _+_

```
mod! NAT* {
 pr(NAT+)
  ...
 eq N *   0 = 0  .
 eq M * s N = M + (M * N). }
```

$F_1 =$ _*_

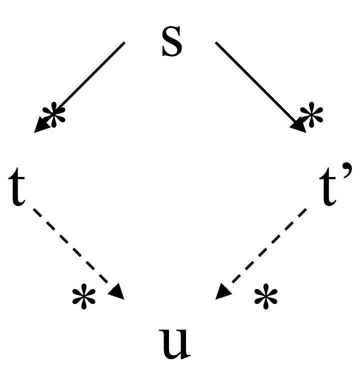
Recursive Path Order

- RPO is a well-founded partial order on terms, defined from a given precedence order on operators
  - Termination of a hierarchical designed specification can be proved by RPO with the precedence of $F_i > F_{i-1} > F_{i-2} > \ldots > F_1 > F_0 > C$
  - For a specification whose termination cannot be proved by RPO, you may find useful powerful termination provers: AProVE, C*i*ME, TTT, etc

Confluence

Definition of Confluence

- SP is confluent if 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




```
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))

0
```

Termination and Confluence

- Confluence guarantees that a normal form of a given term is unique
- A terminating and confluent SP gives us sound and complete equational reasoning:
  - Reduce both sides of a given equation
  - Compare their normal forms
    - If they are same, the equation is deducible from the axiom
    - If they are not same, it is not

Branch

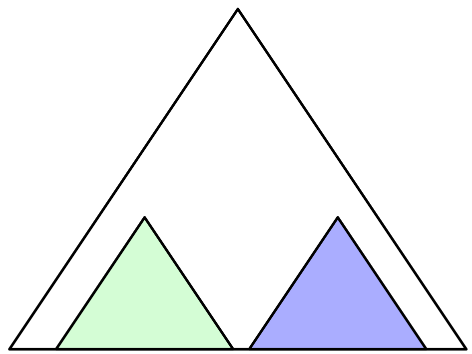
- If some operator has more than one arities, it may include more than one redexes

  f(b, a) <-- f(a, a) --> f(a, b)   (by eq a = b .)

- Such branches can be recovered by rewriting each other redex of the previous rewriting

  f(b, a) --> f(b, b) <-- f(a, b)

- What branches are troublesome?



Overlap

- Terms overlap if a one’s instance is an instance of the other’s non-variable subterm
  - A branch resulting from an overlap may not be recovered because a redex may disappear

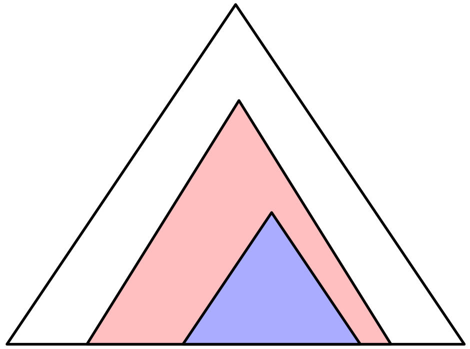
```
eq (X + Y) + Z = X +(Y + Z) .
eq first(X + Y) = X .
```

first((0 + s 0) + s s 0)

first(0 + (s 0 + s s 0))

0 + s 0

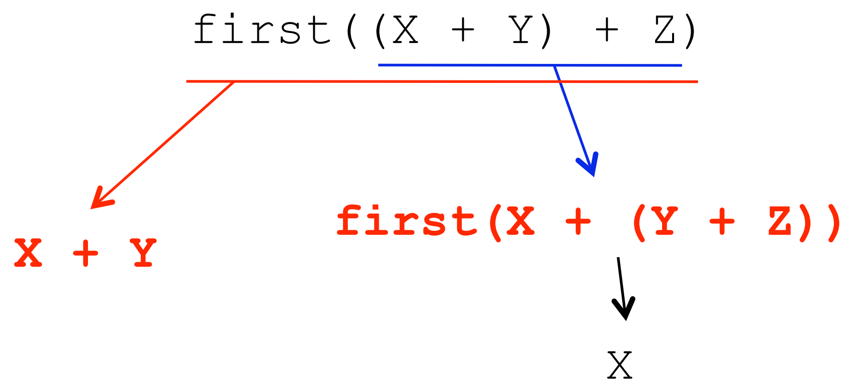
0



Critical Pair

- The most general unifier of overlaps of rewrite rules has two direct descendants.
  - The pair is called a critical pair

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



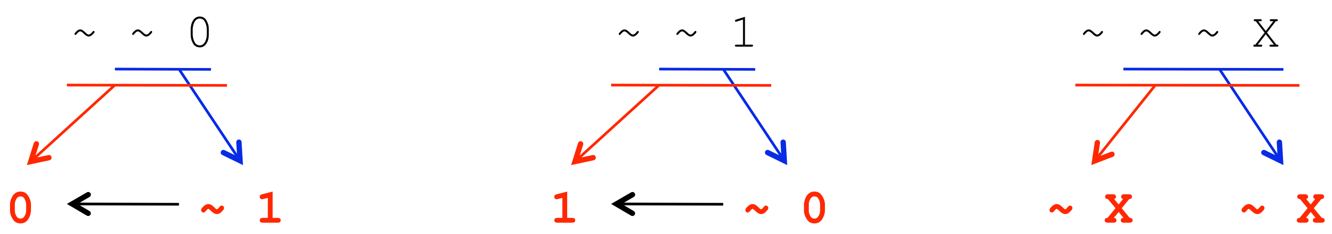
- The CP (**X + Y**, **first(X + (Y + Z))**) is not joinable, and this SP is not confluent

Sufficient condition of Confluence

- **[Theorem]** (Knuth and Bendix 1970): If SP is terminating and all critical pairs are joinable, then SP is confluent
  - [Example] `BOOL-NOT` has three CPs: (0, ~1), (1, ~0) and (~ X, ~X), and all those CPs are joinable, thus, it is confluent

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

BOOL-NOT



How to write confluent SP

- No overlap implies no critical pair
  - `NAT+` and `NAT*` are confluent since their rewrite rules do not overlap and there is no CP
- If there are CPs, check the joinnability of all critical pairs
  - It is easy to make CPs if every lhs is in the form of $F_i(t_1, \ldots t_n)$ where t_i are constructor terms

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

All CPs are joinable:

```
( 0, 0 )
( s N, s (0 + N) )
( s M, s (M + 0) )
```

Sufficient completeness

Definition of Sufficient completeness

- A specification is sufficiently complete if for every ground (variable-free) term ***t***, there exists a constructor term ***t'*** such that ***t*** = ***t'*** can be deduced from the axiom
  - Roughly speaking, every non-constructor operator F_i is defined for all constructor terms

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

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

Operator _+_ is not defined for (`s 0`, `0`), and there is no constructor term ***t*** such that `s 0 + 0` $=_E$ ***t***

Consistency

- If SP-F is declared with the initial denotation (mod!), F should be defined for all patterns, i.e. SP-F should be sufficiently complete
- If not so, SP-F denotes no model (inconsistent !)

```
mod! NAT+ {
 pr(BASIC-NAT) ...
 eq N +    0 = N
 eq M + s N = s (M + N) .
}
```

```
mod! NAT+ {
 pr(BASIC-NAT) ...
 eq 0 +    N = N
 eq M + s N = s (M + N) .
}
```

Sufficient condition for Sufficient completeness

- [Proposition] If SP is terminating and satisfies the following condition, then SP is sufficiently complete
  - For any non-constructor operator F_i and constructor terms $t_1, \ldots, t_n$, there exists an equation $F_i(l_1, \ldots l_n) = r$ applicable to the term $F_i(t_1, \ldots t_n)$ (that is, $F_i(t_1, \ldots t_n)$ is an instance of $F_i(l_1, \ldots l_n)$)

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

Term `s`^{n} `0 + s`^{m} `0` is an instance of `N + 0` (when n = 0)
or an instance of `M + s N` (when n > 0)

Into a single module

- If all SP-F_i are sufficiently complete, BASIC-F and SP-F_i can be written into a single module without changing their models
  - In the following lectures, you may find such modules

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

```
mod! NAT+ {...
 eq N +   0 = N .
 eq M + s N = s (M + N)  . }
```

```
mod! NAT* {...
 eq N *   0 = 0 .
 eq M * s N = M + (M * N). }
```

```
mod! NAT* {
 [Zero NzNat < Nat]
 op 0  : -> Zero
 op s_ : Nat -> NzNat
 ops (_+_) (_*_) : ...
 eq N +   0 = N .
 eq M + s N = s (M + N)  .
 eq N *   0 = 0 .
 eq M * s N = M + (M * N).
}
```

Conditional Equations

Conditional equations

- CafeOBJ allows us to write a condition for an equation
  - A condition is a term of Boolean sort `Bool`

NAT-EVEN

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

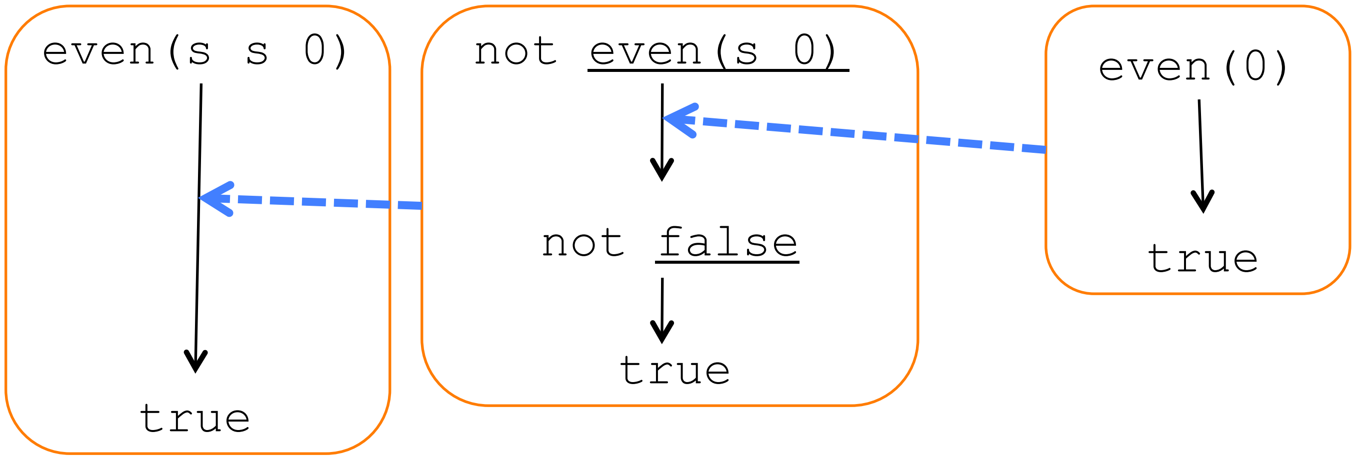
- CafeOBJ modules import a built-in Boolean module `BOOL` implicitly, thus, you can use Boolean operators to write conditional equations without any explicit import of `BOOL`

Reduction by conditional equations

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

NAT-EVEN

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



Termination for conditional SP

- 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 for conditional SP

- Most of conditional SPs overlap, because conditions are used to write case-splitting for 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)` and `not P(X)`
  - `X < 5,  5 <= X and X < 10` and `10 <= X`

Sufficient completeness for conditional SP

- For sufficient completeness, conditions of a pattern should cover all cases, i.e., $c_0 \lor c_1 \lor \ldots \lor c_n$ = `true`

NAT-EVEN

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

  - In order to obtain confluent and sufficiently complete SP, for each instance F(t) of a pattern F(l), there exists a **unique** conditional equation such that the condition is `true`
    - `X < 5,  7 <= X` (not sufficient completeness)
    - `X < 5,  3 <= X` (may not be confluent)

Associative Commutative Operators

Associative Commutative operators

- (Explicit) Equations for associative and commutative laws may cause non-termination or 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 }
```

[Merit] For associative operators, some brackets can be omitted.
For example, CafeOBJ can parse `0 + s 0 + s s 0 + s s s 0 .`

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 terms of `Elt` is a term of `Bag`
  e.g. `a b a b c` 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 Rewriting $\texttt{-->}_{AC}$ is defined as the composition ($=_{AC}$ o $\texttt{-->}$)

```
eq E in (E B) = true .
```

```
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, check all AC equivalent terms, and if there is a redex, 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 seems to be terminating, AC attribute may make 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 seems to be confluent, AC attribute may make it non-confluent

BAG3

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

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

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

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

Execise

- Write specifications of the following operators on natural numbers:
  - Subtraction operation: `_-_ : Nat Nat -> Nat`
  - Greater-than operation: `_>_ : Nat Nat -> Bool`
  - Modulo operation: `_mod_ : Nat Nat -> Nat`
  - GCD(Greatest Common Measure) operation: `gcd: Nat Nat -> Nat`