Reasoning by Rewriting 1

CafeOBJ Team of JAIST

Topics

♦ The execution of CafeOBJ specification, which is based on the rewriting (or TRS: Term Rewriting System), is introduced.

♦ Termination of execution of CafeOBJ specification is explained by termination of TRS, and a method to write terminating CafeOBJ specification is discussed.
An equational proof

- Consider a problem that $s(0) + 0$ and $0 + s(0)$ are equal or not w.r.t. the specification $\text{NAT}+2$
  - One may try to find an equational connection between them
    1. $s(0) + 0 = s(0)$ from the 1st equation
    2. $s(0) = s(0 + 0)$ from the 1st eq. reversely
    3. $s(0 + 0) = 0 + s(0)$ from the 2nd eq. reversely
    4. $s(0) + 0 = 0 + s(0)$ from Transitivity, 1, 2, 3
- Constructing this kind of proofs is not suited for automation

Rewrite rules and Rewriting

- The Term Rewriting System (TRS) gives us an efficient way to prove equations.
- In TRS, a (bidirectional) equation is regarded as a (left-to-right) rewrite rule.
- An instance of the left-hand side (lhs) of a rewrite rule is called a redex.
- Rewriting is to replace a redex into the corresponding instance of the right-hand side (rhs).
  - For example, $s(0+s(0)) \rightarrow s(s(0+0))$
Equational Reasoning by Reduction (1)

- **Reducing** a term is to repeat the rewriting until it cannot be done, that is, there is no redex.
  - For example,
    \[ s(0 + s(0)) \rightarrow s(s(0 + 0)) \rightarrow s(s(0)) \]
  - A term with no redex is called a normal form
- Equational reasoning by reduction is to reduce given two terms into their normal forms. If the normal forms are same, then the two terms are proved to be equal.
  - For example,
    \[ 0 + s(0) \rightarrow* s(0) \text{ and} \]
    \[ s(0) + 0 \rightarrow* s(0), \text{ hence} \]
    \[ 0 + s(0) = s(0) + 0 \text{ holds} \]

\[\begin{align*}
\text{mod! } \text{NAT}+2 & \{ \ldots \\
\text{eq } N + 0 & = N \\
\text{eq } M + s(N) & = s(M + N) .\}
\end{align*}\]

Equational Reasoning by Reduction (2)

- Equational Reasoning (n can be large number)
- Reduction
  - \( s \)
  - \( t_1 \)
  - \( t_2 \)
  - \( \ldots \)
  - \( t_n \)
  - \( u \)
  - \( u_1 \)
  - \( u_2 \)
  - \( \ldots \)
  - \( u_{n+1} \)
CafeOBJ reduction command

CafeOBJ supports reduction based on TRS

CafeOBJ> reduce in NAT+2 : 0 + s(0) == s(0) + 0.
-- reduce in NAT+2 : 0 + s(0) == s(0) + 0
(true):Bool

CafeOBJ> set trace whole on
CafeOBJ> reduce in NAT+2 : 0 + s(0) == s(0) + 0.
-- reduce in NAT+ : 0 + s(0) == s(0) + 0
[1]: 0 + s(0) == s(0) + 0 ---> s(0 + 0) == s(0) + 0
[2]: s(0 + 0) == s(0) + 0 ---> s(0) == s(0) + 0
[3]: s(0) == s(0) + 0 ---> s(0) == s(0)
[4]: s(0) == s(0) ---> true
(true):Bool

mod! NAT+2 { ...
eq N + 0 = N .
eq M + s(N) = s(M + N) .}

Construct a proof from a trace

Rewriting implies equality, that is, \( t = t' \) if \( t \rightarrow t' \)

From a trace of reduction, we can extract a proof of the equality.
- Get the traces of reduction for both lhs and rhs, and
- Construct a proof

CafeOBJ> red in NAT+2: 0 + s(0)
[1]: 0 + s(0) ---> s(0 + 0)
[2]: s(0 + 0) ---> s(0)

CafeOBJ> red in NAT+2: s(0) + 0
[1]: s(0) + 0 ---> s(0)

A PROOF OF “\( 0 + s(0) = s(0) + 0 \)”
[1]: \( 0 + s(0) = s(0 + 0) \) from the 2nd eq.
[2]: \( s(0 + 0) = s(0) \) from the 1st eq.
[3]: \( s(0) + 0 = s(0) \) from the 1st eq.
From Transitivity, L1, L2, and R1, \( 0 + s(0) = s(0) + 0 \) holds
Trace of CafeOBJ reductions

```plaintext
%NAT+2 + EQL> -- reduce in %NAT+2 + EQL : ((s(0) + 0) = (0 + s(0))):Bool
1>[1] rule: eq (N:Nat + 0) = N
   ( N:Nat |-> s(0) )
1<[1] (s(0) + 0) --> s(0)
[1] ((s(0) + 0) = (0 + s(0))) ---> (s(0) = (0 + s(0)))
   ( M:Nat |-> 0, N:Nat |-> 0 )
1<[2] (0 + s(0)) --> s((0 + 0))
[2] ( (s(0) = (0 + s(0))) ---> (s(0) = s((0 + 0)))
1>[3] rule: eq (N:Nat + 0) = N
   ( N:Nat |-> 0 )
1<[3] (0 + 0) --> 0
[3] ( (s(0) = s((0 + 0))) ---> (s(0) = s(0))
1>[4] rule: eq (CUX = CUX) = true
   ( CUX |-> s(0) )
1<[4] (s(0) = s(0)) --> true
[4] ( (s(0) = s(0)) ---> true
1<[4] (true) :Bool
(0.000 sec for parse, 4 rewrites(0.000 sec), 5 matches)
```

More on implementation

♦ To implement a reduction, we need to choose the following things:
  • Choose a redex: A term may have more than one redexes
  • Choose an equation: A redex may be rewritten by more than one equations

```
eq a = b .
eq c = d .
eq c = e .
f(a, c) --> f(b, c)
f(a, c) --> f(a, d)
f(a, c) --> f(a, e)
```

♦ In CafeOBJ,
  • The Evaluation strategy (E-strategy) decide one redex from a term
  • Equations are applied in order to top-to-bottom
Evaluation strategy

- The Evaluation strategy is a user-defined strategy, which is given as operators attributes, like
  \[
  \text{op } _+_- : \text{Int Int } \rightarrow \text{Int } \{ \text{strat: (1 2 0)} \}
  \]
- A term \( t_1 + t_2 \) is reduced as follows:
  - Reduce \( t_1 \) into \( t_1' \)
  - Reduce \( t_2 \) into \( t_2' \)
  - If \( t_1' + t_2' \) is a redex then rewrite it, and reduce again
  - Otherwise, return \( t_1' + t_2' \)
- CafeOBJ automatically gives a suitable local strategy for each operator when loading a specification. So, basically you do not care about it

An Implementation is not the Model of Specification

- Do not confuse the model of a specification and the implementation of reduction
  - TRS or the reduction command is not a model of SP.
    Just a support tool for equational reasoning

-- A Bad Example
mod! ZERO{...
op zero : \text{Nat } \rightarrow \text{Bool}
  eq zero(0) = true .
  eq zero(N: Nat) = false .
}

\[
\begin{align*}
\text{ZERO> red zero(0) .} \\
\text{(true): Bool} \\
\text{ZERO> red zero(s(s(0))) .} \\
\text{(false): Bool}
\end{align*}
\]
- In reduction, \( \text{zero}(n) \) returns true when \( n = 0 \), otherwise false, BUT,
- In a model, \( \text{zero}(n) = \text{false} \) for any \( n \), including 0. So, \( \text{true} = \text{zero}(0) = \text{false} \), and ZERO is inconsistent!
Properties of TRS

- TRS is a powerful tool to prove equations, however, it does not always behave well, because
  - It may not terminate
  - It may not return true even if the truth can be gotten by reasoning with equations
- To write a specification in which the reduction command works well, we need to study fundamental properties of TRS:
  - Termination,
  - Confluence,
  - Sufficient completeness, etc.
**Notations**

- \( t \rightarrow t' \) : One-step rewriting
- \( t \rightarrow^* t' \) : Many-steps rewriting, including zero step
- \( t \leftarrow \rightarrow^* \rightarrow t' \) : divided; \( s \rightarrow^* t \) and \( s \rightarrow^* t' \) for some \( s \)

\[
\begin{align*}
(0 + 0) & \rightarrow^* 0 + s(0) \\
(0 + 0) + s(0) & \rightarrow s((0 + 0) + 0) \rightarrow s(0 + 0) \\
(0 + 0) + s(0) & \rightarrow 0 + s(0)
\end{align*}
\]

- \( t \leftarrow\rightarrow^* \rightarrow t' \) : Joinable; \( t \rightarrow^* u \) and \( t' \rightarrow^* u \) for some \( u \)

\[
\begin{align*}
s(0 + 0) & \rightarrow^* \rightarrow 0 + s(0) \\
s(0 + 0) & \rightarrow s(0) \\
0 + s(0) & \rightarrow s(0 + 0) \rightarrow s(0)
\end{align*}
\]

**Necessary conditions of term rewriting system**

- In order to be a rewrite rule, an equation should satisfy the following conditions about variables occurrences:
  - **Condition 1**: Any left-hand side is not a variable:
    - If such an equation exists, any term can be rewritten and get into infinite rewriting loop
      
      \[
      \text{eq } X = 0 \ . \quad 0 \rightarrow 0 \rightarrow 0 \rightarrow \ldots
      \]
  
  - **Condition 2**: Any variable in a rhd appears in the lhs:
    - Such a variable is called an **extra-variable**.
    - If an extra-variable exists, we need to give an information to instantiate the variable
      
      \[
      \text{eq } f(X) = X + Y \ . \quad f(0) \rightarrow 0 + ??
      \]
Ill-defined rewrite rules ignored

\[
\text{mod! TEST} \{
\begin{array}{l}
[\text{Elt}]
\text{ops } 0\ 1 : \to \text{Elt}
\text{var } X : \text{Elt}
\text{eq } X = 0 .
\end{array}
\}
\text{red in TEST} : 1 .
\]

\[
\text{mod! TEST2} \{
\begin{array}{l}
[\text{Elt}]
\text{op } 0 : \to \text{Elt}
\text{op } f : \text{Elt} \to \text{Elt}
\text{op } _+ : \text{Elt} \text{ Elt} \to \text{Elt}
\text{var } X\ Y : \text{Elt}
\text{eq } f(X) = X + Y .
\end{array}
\}
\text{red in TEST2} : f(0) .
\]

Those equations which are not rewrite rules are ignored in the reduction

Termination
Termination

Definition: A specification SP is terminating iff any infinite sequence \( t_0 \rightarrow t_1 \rightarrow \ldots \) does not exists.

- \( \text{NAT+COM} \) is not terminating:

\[
\text{mod! NAT+COM}
\begin{align*}
\text{eq } & N + M = M + N . \\
\text{}}
\]

\( s(0) + 0 \rightarrow 0 + s(0) \rightarrow s(0) + 0 \rightarrow \ldots \)

- Termination guarantees that any term has its normal form, and makes us possible to compute a normal form in finite times

Decreasing something

Roughly speaking, SP is terminating if any rewrite decreases something.

Example: EVEN defines a predicate to check even numbers. The size of a term decreases for any equation. So, EVEN is terminating.

\[
\text{mod! EVEN}
\begin{align*}
\text{eq } [1] : \text{even}(0) &= \text{true} . \\
\text{eq } [2] : \text{even}(s(0)) &= \text{false} . \\
\text{eq } [3] : \text{even}(s(s(N))) &= \text{even}(N) . \\
\text{}}
\]
Recursively defined functions

♦ How about $\text{NAT}+2$?

\[
\text{mod! NAT}+2 \{ \quad \ldots \\
\text{eq } N + 0 = N \\
\text{eq } M + s(N) = s(M + N) \} .
\]

♦ The 2nd argument decreases for each rewrite.

\[
0 + s(s(0))) \rightarrow s(0 + s(0))) \rightarrow s(s(0 + 0))) \rightarrow s(s(s(0)))
\]

♦ $(+_\text{nat+2})$ is a so-called recursively defined function

♦ If each function is recursively defined, SP is terminating

Declaring domain and co-domain (rang) of a function as sets of constructor terms

♦ Write a BASIC-SP for a function’s domain and range
  - An operator in BASIC-SP is called a constructor. The set of them is denoted by $C$
  - Constructor terms, which are constructed by constructors, denote elements of the domain, The set is $T_C$

\[
\text{mod! BASIC-NAT} \{ \quad \ldots \\
\text{op } 0 : \rightarrow \text{Zero} \{\text{constr}\}
\text{op } s : \text{Nat} \rightarrow \text{NzNat} \{\text{constr}\}
\}
\]

\[
C = \{0, s\}
T_C = \{0, s(0), s(s(0)), \ldots\}
\]
Hierarchical Definition of functions

♦ Write a SP-F for a function F with importing BASIC-SP

```
mod! NAT+2{
  pr(BASIC-NAT)
  op __+ : Nat Nat -> Nat
  eq N + 0 = N .
  eq M + s(N) = s(M + N) .
}
```

♦ Write a SP-G for a function G which is defined by using F with importing SP-F

```
mod! NAT*2{
  pr(NAT+)
  op __* : Nat Nat -> Nat
  eq M * 0 = 0 .
  eq M * s(N) = M + (M * N) .
}
```

How to write a recursively defined function (1)

♦ The right-hand sides should be constructed from variables, constructors, pre-defined functions and recursive calls (with smaller argument)
  • \( F(\ldots,t,\ldots) \) is a recursive call of \( F(\ldots,t',\ldots) \) iff \( t \) is a subterm of \( t' \)

```
mod! EVEN{ ...
  eq even(0)       = true .
  eq even(s(0))    = false .
  eq even(s(s(N))) = even(N) .
}
```

```
mod! NAT+2{ ...
  eq N + 0 = N .
  eq M + s(N) = s(M + N) .
}
```
How to write a recursively defined function (2)

♦ The right-hand sides should be constructed from variables, constructors, pre-defined functions and recursive calls (with smaller argument)
  • \( F(\ldots,t,\ldots) \) is a recursive call of \( F(\ldots,t',\ldots) \) iff \( t \) is a subterm of \( t' \)

\[
\text{mod! NAT*2} \{ \text{pr}(\text{NAT+}) \ldots \\
\text{eq } N \ast 0 = 0 . \\
\text{eq } M \ast s(N) = M + (M \ast N). \}
\]

\[
\text{mod! NAT-FACT} \{ \text{pr}(\text{NAT*}) \ldots \\
\text{eq } \text{fact}(0) = s(0) . \\
\text{eq } \text{fact}(s(N)) = s(N) \ast \text{fact}(N). \}
\]

How to write terminating specifications

♦ If all functions are defined recursively in the manner described, SP is terminating
♦ Recursive Path Ordering (RPO), one of the most famous termination proving techniques, proves its termination
♦ If you want to write a function which cannot be written as a recursively defined one, other termination provers may help you
  • Dependency pair techniques
  • AProVE, CiME, TTT, etc