

Introduction to Formal Methods

Min Zhang

zhangmin@sei.ecnu.edu.cn

2020-2021/Spring



Software Engineering Institute

Software Engineering Institute
East China Normal University

Min Zhang

Introduction to Formal Methods

Contents of the lecture

Lecture 01: Introduction to Formal Methods

Lecture 02: Introduction about Maude

Lecture 03: Functional Module

Lecture 04: System Module

Lecture 05: Searching & Model Checking

Lecture 06: Reflection and Meta-Programming in Maude

Lecture 07: Implementation of a mini programming language in Maude

Lecture 08: Mathematical Foundation of Maude: Term Rewriting

Lecture 01: Introduction to Formal Methods

Welcome to the world of Formal Methods



About Formal Methods

In computer science, specifically software engineering and hardware engineering, formal methods are a particular kind of **mathematically** based techniques for the **specification, development and verification** of software and hardware systems.

About Formal Methods

非形式化验证



系统/代码/算法/协议

是否安全/可信



性質



形式化验证

建模

$$f(x) = \sum_{i=0}^{\infty} \frac{f^{(i)}(0)}{i!} x^i$$

$$e^{-mx^2} = \frac{1}{\sqrt{\pi}} \int_0^{\infty} e^{-t^2} t^{m-1} dt$$

$$\sin mx = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

$$\operatorname{TRIGONOMETER}$$

$$\begin{aligned} f(x) &= \sum_{i=0}^{\infty} \frac{f^{(i)}(0)}{i!} x^i \\ &= \sum_{i=0}^{\infty} \frac{f^{(i)}(0)}{i!} (bx)^i \\ &= b^i \sum_{i=0}^{\infty} \frac{f^{(i)}(0)}{i!} x^i \\ &= b^i f(x) \end{aligned}$$

系统模型

定之

$$E=mc^2$$

逻辑公式



Formal Methods in Computer Science



Turing Award Winners in Formal Methods

More than 12 Turing Award winners for their work on formal methods.



Figure: Pnueli was born in Nahalal, in the British Mandate of Palestine (now in Israel) and received a Bachelor's degree in mathematics from the Technion in Haifa, and Ph.D. in applied mathematics from the Weizmann Institute of Science. His thesis was on the topic of "Calculation of Tides in the Ocean". He switched to computer science during a stint as a post-doctoral fellow at Stanford University. His works in computer science focused on temporal logic and model checking, particularly regarding fairness properties of concurrent systems

Turing Award Winners in Formal Methods

More than 12 Turing Award winners for their work on formal methods.

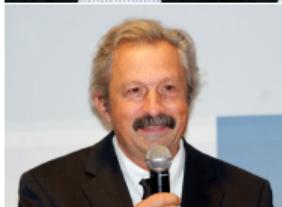
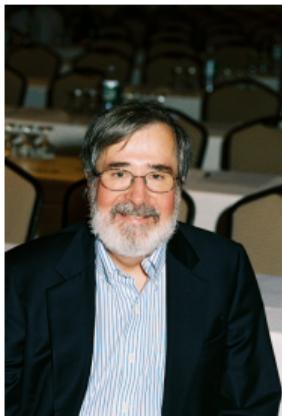


Figure: Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis.
For their roles in developing model checking into a highly effective verification technology, widely adopted in the hardware and software industries

Turing Award Winners in Formal Methods

More than 12 Turing Award winners for their work on formal methods.

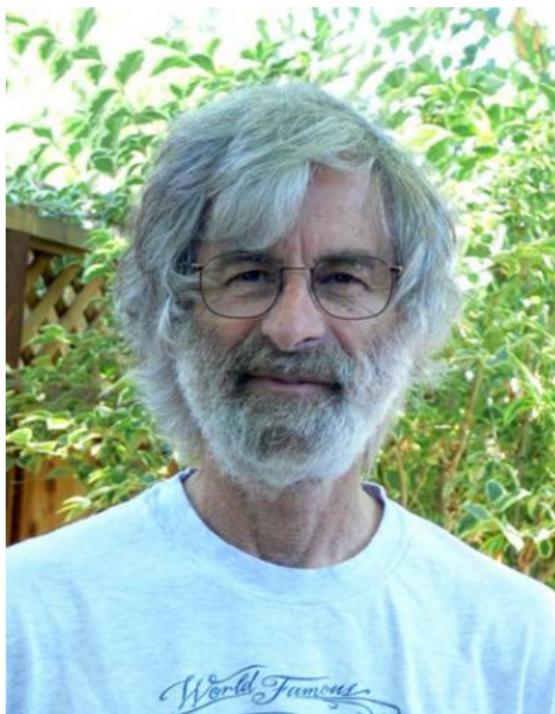


Figure: Leslie B. Lamport is an American computer scientist. Lamport is best known for his seminal work in distributed systems, and as the initial developer of the document preparation system LaTeX and the author of its first manual. Leslie Lamport was the winner of the 2013 Turing Award for imposing clear, well-defined coherence on the seemingly chaotic behavior of distributed computing systems, in which several autonomous computers communicate with each other by passing messages. He devised important algorithms and developed formal modeling and verification protocols that improve the quality of real distributed systems. These contributions have resulted in improved correctness, performance, and reliability of computer systems

How to do formal methods?

$$\mathcal{M} \models \psi$$

Step 1 : Formalize your target system to be a mathematics model \mathcal{M} ;

How to do formal methods?

$$\mathcal{M} \models \psi$$

Step 1 : Formalize your target system to be a mathematics model \mathcal{M} ;

Step 2 : Formalize the properties you want to prove to be a formula ψ ;

How to do formal methods?

$$\mathcal{M} \models \psi$$

Step 1 : Formalize your target system to be a mathematics model \mathcal{M} ;

Step 2 : Formalize the properties you want to prove to be a formula ψ ;

Step 3 : Study the **decidability** of $\mathcal{M} \models \psi$;

How to do formal methods?

$$\mathcal{M} \models \psi$$

Step 1 : Formalize your target system to be a mathematics model \mathcal{M} ;

Step 2 : Formalize the properties you want to prove to be a formula ψ ;

Step 3 : Study the **decidability** of $\mathcal{M} \models \psi$;

Step 4 : If undecidable, simplify it. If decidable, study its **complexity**;

How to do formal methods?

$$\mathcal{M} \models \psi$$

Step 1 : Formalize your target system to be a mathematics model \mathcal{M} ;

Step 2 : Formalize the properties you want to prove to be a formula ψ ;

Step 3 : Study the **decidability** of $\mathcal{M} \models \psi$;

Step 4 : If undecidable, simplify it. If decidable, study its **complexity**;

Step 5 : Devise an (efficient hopefully) algorithm to check $\mathcal{M} \models \psi$;

How to do formal methods?

$$\mathcal{M} \models \psi$$

Step 1 : Formalize your target system to be a mathematics model \mathcal{M} ;

Step 2 : Formalize the properties you want to prove to be a formula ψ ;

Step 3 : Study the **decidability** of $\mathcal{M} \models \psi$;

Step 4 : If undecidable, simplify it. If decidable, study its **complexity**;

Step 5 : Devise an (efficient hopefully) algorithm to check $\mathcal{M} \models \psi$;

Step 6 : Implement a prototype tool;

How to do formal methods?

$$\mathcal{M} \models \psi$$

Step 1 : Formalize your target system to be a mathematics model \mathcal{M} ;

Step 2 : Formalize the properties you want to prove to be a formula ψ ;

Step 3 : Study the **decidability** of $\mathcal{M} \models \psi$;

Step 4 : If undecidable, simplify it. If decidable, study its **complexity**;

Step 5 : Devise an (efficient hopefully) algorithm to check $\mathcal{M} \models \psi$;

Step 6 : Implement a prototype tool;

Step 7 : Do experiment;

How to do formal methods?

$$\mathcal{M} \models \psi$$

Step 1 : Formalize your target system to be a mathematics model \mathcal{M} ;

Step 2 : Formalize the properties you want to prove to be a formula ψ ;

Step 3 : Study the **decidability** of $\mathcal{M} \models \psi$;

Step 4 : If undecidable, simplify it. If decidable, study its **complexity**;

Step 5 : Devise an (efficient hopefully) algorithm to check $\mathcal{M} \models \psi$;

Step 6 : Implement a prototype tool;

Step 7 : Do experiment;

Step 8 : Win a Turing Award (with luck).

How to learn formal methods

$$\mathcal{M} \models \psi$$

Level 1 (master):

Step 1 : Learn to model in some modeling language;

How to learn formal methods

$$\mathcal{M} \models \psi$$

Level 1 (master):

Step 1 : Learn to model in some modeling language;

Step 2 : Learn to formulate properties in some logic, LTL, CTL, etc.

How to learn formal methods

$$\mathcal{M} \models \psi$$

Level 1 (master):

Step 1 : Learn to model in some modeling language;

Step 2 : Learn to formulate properties in some logic, LTL, CTL, etc.

Step 3 : Learn an existing formal verification tools, SPIN, Coq, NuXmv, Uppaal;

How to learn formal methods

$$\mathcal{M} \models \psi$$

Level 1 (master):

Step 1 : Learn to model in some modeling language;

Step 2 : Learn to formulate properties in some logic, LTL, CTL, etc.

Step 3 : Learn an existing formal verification tools, SPIN, Coq, NuXmv, Uppaal;

How to learn formal methods

$$\mathcal{M} \models \psi$$

Level 1 (master):

Step 1 : Learn to model in some modeling language;

Step 2 : Learn to formulate properties in some logic, LTL, CTL, etc.

Step 3 : Learn an existing formal verification tools, SPIN, Coq, NuXmv, Uppaal;

Level 2 (PhD, professor):

Step 1 : Design a modeling language to a general/specific class of systems;

How to learn formal methods

$$\mathcal{M} \models \psi$$

Level 1 (master):

Step 1 : Learn to model in some modeling language;

Step 2 : Learn to formulate properties in some logic, LTL, CTL, etc.

Step 3 : Learn an existing formal verification tools, SPIN, Coq, NuXmv, Uppaal;

Level 2 (PhD, professor):

Step 1 : Design a modeling language to a general/specific class of systems;

Step 2 : Design a logic for target properties.

How to learn formal methods

$$\mathcal{M} \models \psi$$

Level 1 (master):

Step 1 : Learn to model in some modeling language;

Step 2 : Learn to formulate properties in some logic, LTL, CTL, etc.

Step 3 : Learn an existing formal verification tools, SPIN, Coq, NuXmv, Uppaal;

Level 2 (PhD, professor):

Step 1 : Design a modeling language to a general/specific class of systems;

Step 2 : Design a logic for target properties.

Step 3 : Develop a tool to support the modeling language and property verification.

Teaching Formal Methods

Courses and books:

- formal methods in pure academia:
[Open resources on formal methods in education](#)

- formal methods for industrial application:
[Moocs, Blogs, Tools, Projects, etc.](#)

References

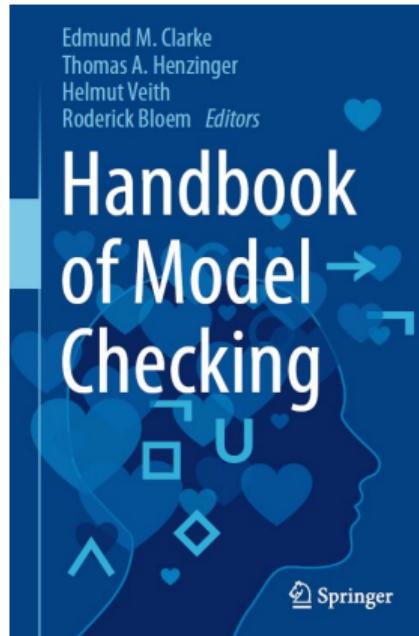


Figure: The editors and authors of this handbook are among the world's leading researchers in this domain, and the 32 contributed chapters present a thorough view of the origin, theory, and application of model checking. In particular, the editors classify the advances in this domain and the chapters of the handbook in terms of two recurrent themes that have driven much of the research agenda: the algorithmic challenge, that is, designing model-checking algorithms that scale to real-life problems; and the modeling challenge, that is, extending the formalism beyond Kripke structures and temporal logic.

See a [complete list](#) of references.

Two paradigms in formal methods

1 Theorem proving: to **prove** properties hold in systems.

What is **proof**?

A formal proof or derivation is a finite sequence of sentences (called well-formed formulas in the case of a formal language), each of which is an axiom, an assumption, or follows from the preceding sentences in the sequence by a rule of inference.

2 Model checking: to **check** properties hold in systems.

What is **check**?

Enumerate all possible cases and see if a property holds in all cases.

Objectives of this lecture

- 1 A general-purpose model language: **Maude**

Objectives of this lecture

- 1 A general-purpose model language: **Maude**
- 2 Ideas behind Maude

Objectives of this lecture

- 1 A general-purpose model language: **Maude**
- 2 Ideas behind Maude
 - What is computation

Objectives of this lecture

1 A general-purpose model language: **Maude**

2 Ideas behind Maude

- What is computation
- What is modeling

Objectives of this lecture

1 A general-purpose model language: **Maude**

2 Ideas behind Maude

- What is computation
- What is modeling
- What is reasoning

Objectives of this lecture

1 A general-purpose model language: **Maude**

2 Ideas behind Maude

- What is computation
- What is modeling
- What is reasoning

3 How to do model checking using Maude

Objectives of this lecture

1 A general-purpose model language: **Maude**

2 Ideas behind Maude

- What is computation
- What is modeling
- What is reasoning

3 How to do model checking using Maude

4 How to do theorem proving using Maude

Objectives of this lecture

- 1 A general-purpose model language: **Maude**
- 2 Ideas behind Maude
 - What is computation
 - What is modeling
 - What is reasoning
- 3 How to do model checking using Maude
- 4 How to do theorem proving using Maude
- 5 How to implement your modeling language and verification tool using Maude

Textbook and resources

For practical purpose:

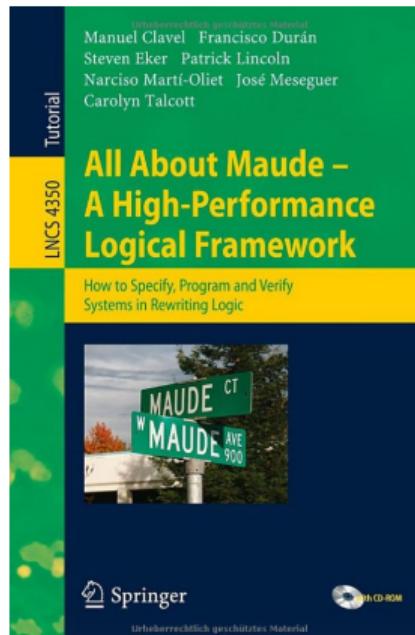


Figure: The Maude system is an implementation of rewriting logic developed at SRI International. It is similar in its general approach to Joseph Goguen's OBJ3 implementation of equational logic, but based on rewriting logic rather than order-sorted equational logic, and with a heavy emphasis on powerful metaprogramming based on reflection.

Textbook and resources

For theoretical purpose:

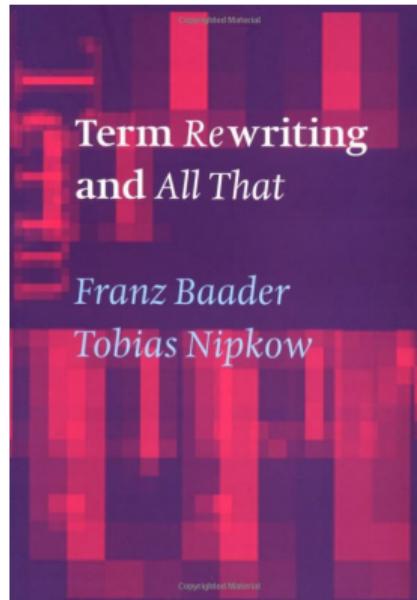


Figure: In mathematics, computer science, and logic, rewriting covers a wide range of (potentially non-deterministic) methods of replacing subterms of a formula with other terms. The objects of focus for this article include rewriting systems (also known as rewrite systems, rewrite engines or reduction systems). In their most basic form, they consist of a set of objects, plus relations on how to transform those objects.

People you need to know



Figure: Joseph Goguen was a US computer scientist. He was professor of Computer Science at the University of California and University of Oxford and held research positions at IBM and SRI International. Goguen's work was one of the earliest approaches to the algebraic characterisation of abstract data types and he originated and helped develop the **OBJ** family of programming languages. He was author of *A Categorical Manifesto* and founder and Editor-in-Chief of the Journal of Consciousness Studies. His development of institution theory impacted the field of universal logic. Standard implication in product fuzzy logic is often called "Goguen implication". Goguen categories are named after him.

People you need to know



Figure: José Meseguer is Professor of Computer Science at UIUC and leads the Formal Methods and Declarative Languages Laboratory. He obtained his Ph.D. in **Mathematics** at the University of Zaragoza, Spain, in 1975. After post-doctoral stays at the University of Santiago de Compostela, and at the University of California at Berkeley, he joined in 1980 the Computer Science Laboratory at SRI International in Menlo Park, California, where he became a Principal Scientist and Head of the Logic and Declarative Languages Group. He joined the University of Illinois at Urbana-Champaign in 2001. He has worked on the design and implementation of several declarative languages, including the OBJ and Maude languages, on formal specification and verification techniques, on concurrency theory, on formal approaches to object-oriented specification, on parallel software and architectures for declarative languages, and on the logical foundations of computer science using equational logic, rewriting logic, and the theory of general logics.

People you need to know



Figure: Grigore Rosu is a professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign (UIUC), where he leads the Formal Systems Laboratory (FSL), and the founder and president of Runtime Verification, Inc (RV). His research interests encompass both theoretical foundations and system development in the areas of formal methods, software engineering and programming languages. Before joining UIUC in 2002, he was a research scientist at NASA Ames. He obtained his Ph.D. at the University of California at San Diego in 2000. Grigore is known for his work on the [K Framework](#)

Advice on learning this lecture

It is not only about teaching and learning for credits. It is **research**.

- How to read scientific papers

Advice on learning this lecture

It is not only about teaching and learning for credits. It is **research**.

- How to read scientific papers
- How to write technical reports

Advice on learning this lecture

It is not only about teaching and learning for credits. It is **research**.

- How to read scientific papers
- How to write technical reports
- How to do presentation

Advice on learning this lecture

It is not only about teaching and learning for credits. It is **research**.

- How to read scientific papers
- How to write technical reports
- How to do presentation
- How to learn yourself

Advice on learning this lecture

It is not only about teaching and learning for credits. It is **research**.

- How to read scientific papers
- How to write technical reports
- How to do presentation
- How to learn yourself
- How to thinking mathematically

Advice on learning this lecture

It is not only about teaching and learning for credits. It is **research**.

- How to read scientific papers
- How to write technical reports
- How to do presentation
- How to learn yourself
- How to thinking mathematically
- How to implement abstract ideas

Get started

Question: How to prove that addition of natural numbers is commutative?

Get started

Question: How to prove that addition of natural numbers is commutative?

To answer this questions:

- what are natural numbers?

Get started

Question: How to prove that addition of natural numbers is commutative?

To answer this questions:

- what are natural numbers?
- what is addition of two natural numbers?

Get started

Question: How to prove that addition of natural numbers is commutative?

To answer this questions:

- what are natural numbers?
- what is addition of two natural numbers?
- what is commutativity?

Get started

Question: How to prove that addition of natural numbers is commutative?

To answer this questions:

- what are natural numbers?
- what is addition of two natural numbers?
- what is commutativity?

Get started

Question: How to prove that addition of natural numbers is commutative?

To answer this questions:

- what are natural numbers?
- what is addition of two natural numbers?
- what is commutativity?

If you have these questions in your mind, you know **mathematical thinking**.

Step 1: Formalizing natural numbers

How?

Step 1: Formalizing natural numbers

How?

Definition (Natural number (Wiki))

In mathematics, the natural numbers are those used for counting (as in "there are six coins on the table") and ordering (as in "this is the third largest city in the country"). In common mathematical terminology, words colloquially used for counting are "cardinal numbers" and words connected to ordering represent "ordinal numbers". The natural numbers can, at times, appear as a convenient set of codes (labels or "names"); that is, as what linguists call nominal numbers, foregoing many or all of the properties of being a number in a mathematical sense.

Giuseppe Peano



Figure: Giuseppe Peano, August 1858 –April 1932) was an Italian mathematician and glottologist. The author of over 200 books and papers, he was a founder of mathematical logic and set theory, to which he contributed much notation. **The standard axiomatization of the natural numbers is named the Peano axioms in his honor.** As part of this effort, he made key contributions to the modern rigorous and systematic treatment of the method of **mathematical induction**.

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set N or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.
- 3 For all natural numbers x and y , if $x = y$, then $y = x$. That is, equality is **symmetric**.

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.
- 3 For all natural numbers x and y , if $x = y$, then $y = x$. That is, equality is **symmetric**.
- 4 For all natural numbers x , y and z , if $x = y$ and $y = z$, then $x = z$. That is, equality is **transitive**.

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.
- 3 For all natural numbers x and y , if $x = y$, then $y = x$. That is, equality is **symmetric**.
- 4 For all natural numbers x , y and z , if $x = y$ and $y = z$, then $x = z$. That is, equality is **transitive**.
- 5 For all a and b , if b is a natural number and $a = b$, then a is also a natural number. That is, the natural numbers are **closed** under equality.

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.
- 3 For all natural numbers x and y , if $x = y$, then $y = x$. That is, equality is **symmetric**.
- 4 For all natural numbers x , y and z , if $x = y$ and $y = z$, then $x = z$. That is, equality is **transitive**.
- 5 For all a and b , if b is a natural number and $a = b$, then a is also a natural number. That is, the natural numbers are **closed** under equality.
- 6 For every natural number n , $s(n)$ is a natural number.

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.
- 3 For all natural numbers x and y , if $x = y$, then $y = x$. That is, equality is **symmetric**.
- 4 For all natural numbers x , y and z , if $x = y$ and $y = z$, then $x = z$. That is, equality is **transitive**.
- 5 For all a and b , if b is a natural number and $a = b$, then a is also a natural number. That is, the natural numbers are **closed** under equality.
- 6 For every natural number n , $s(n)$ is a natural number.
- 7 For all natural numbers m and n , $m = n$ if and only if $s(m) = s(n)$. That is, S is an injection.

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.
- 3 For all natural numbers x and y , if $x = y$, then $y = x$. That is, equality is **symmetric**.
- 4 For all natural numbers x , y and z , if $x = y$ and $y = z$, then $x = z$. That is, equality is **transitive**.
- 5 For all a and b , if b is a natural number and $a = b$, then a is also a natural number. That is, the natural numbers are **closed** under equality.
- 6 For every natural number n , $s(n)$ is a natural number.
- 7 For all natural numbers m and n , $m = n$ if and only if $s(m) = s(n)$. That is, S is an injection.
- 8 If ψ is a unary predicate such that:

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.
- 3 For all natural numbers x and y , if $x = y$, then $y = x$. That is, equality is **symmetric**.
- 4 For all natural numbers x , y and z , if $x = y$ and $y = z$, then $x = z$. That is, equality is **transitive**.
- 5 For all a and b , if b is a natural number and $a = b$, then a is also a natural number. That is, the natural numbers are **closed** under equality.
- 6 For every natural number n , $s(n)$ is a natural number.
- 7 For all natural numbers m and n , $m = n$ if and only if $s(m) = s(n)$. That is, S is an injection.
- 8 If ψ is a unary predicate such that:
 - $\psi(0)$ is true;

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.
- 3 For all natural numbers x and y , if $x = y$, then $y = x$. That is, equality is **symmetric**.
- 4 For all natural numbers x , y and z , if $x = y$ and $y = z$, then $x = z$. That is, equality is **transitive**.
- 5 For all a and b , if b is a natural number and $a = b$, then a is also a natural number. That is, the natural numbers are **closed** under equality.
- 6 For every natural number n , $s(n)$ is a natural number.
- 7 For all natural numbers m and n , $m = n$ if and only if $s(m) = s(n)$. That is, S is an injection.
- 8 If ψ is a unary predicate such that:
 - $\psi(0)$ is true;
 - for every natural number n , $\psi(n)$ implies that $\psi(s(n))$.

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.
- 3 For all natural numbers x and y , if $x = y$, then $y = x$. That is, equality is **symmetric**.
- 4 For all natural numbers x , y and z , if $x = y$ and $y = z$, then $x = z$. That is, equality is **transitive**.
- 5 For all a and b , if b is a natural number and $a = b$, then a is also a natural number. That is, the natural numbers are **closed** under equality.
- 6 For every natural number n , $s(n)$ is a natural number.
- 7 For all natural numbers m and n , $m = n$ if and only if $s(m) = s(n)$. That is, S is an injection.
- 8 If ψ is a unary predicate such that:
 - $\psi(0)$ is true;
 - for every natural number n , $\psi(n)$ implies that $\psi(s(n))$.

Peano Axiom

The Peano axioms define the arithmetical properties of natural numbers, usually represented as a set \mathbb{N} or \mathbb{N} . The non-logical symbols for the axioms consist of a constant symbol 0 and a unary function symbol s .

- 1 0 is a natural number.
- 2 For every natural number x , $x = x$. That is, equality is **reflexive**.
- 3 For all natural numbers x and y , if $x = y$, then $y = x$. That is, equality is **symmetric**.
- 4 For all natural numbers x , y and z , if $x = y$ and $y = z$, then $x = z$. That is, equality is **transitive**.
- 5 For all a and b , if b is a natural number and $a = b$, then a is also a natural number. That is, the natural numbers are **closed** under equality.
- 6 For every natural number n , $s(n)$ is a natural number.
- 7 For all natural numbers m and n , $m = n$ if and only if $s(m) = s(n)$. That is, S is an injection.
- 8 If ψ is a unary predicate such that:
 - $\psi(0)$ is true;
 - for every natural number n , $\psi(n)$ implies that $\psi(s(n))$.

Then, $\psi(n)$ is true for every natural number n .

Step 1: Formalizing natural numbers

```
1 sort Nat .  
2 op 0 : -> Nat .      --- 0 is a constant of Nat  
3 op s : Nat -> Nat . --- s is a constructor
```

Step 2: Formalizing addition

```
1 sort Nat .
2 op _+_ : Nat Nat -> Nat . --- declaration of addition
3 vars X Y : Nat .      --- two variables of Nat
4 eq 0 + X = 0 .      --- axiom
5 eq s(X) + Y = s(X + Y) . --- axiom
```

Step 3: Your homework

Consider how to prove the theorem based on the formulation.

Lecture 02: Introduction about Maude

Install and run Maude

- 1 Download Unix/Mac binary from <http://maude.cs.uiuc.edu>
- 2 Run in terminal `.\maude.linux64`, (Unix) or `.\maude.intelDarwin` (mac)
- 3 Set environment variable `MAUDE_LIB=`your path where `prelude.maude` is located.
Practice: How to set environment variable?
- 4 Set a symbolic link: `sudo ln -s ./maude.linux64 /usr/local/bin/maude`

Program in Maude

- Use your favorite text editor such as Emacs, Vim, GEdit.

- Some important commands of Maude

- **load**: to load a Maude program into the system

- Example: **load pnat.maude**

- **red(reduce)**: to reduce an input to its normal form

- Example: **red s s 0 + s 0 .**

- **parse**: to parse an input

- Example: **parse s s 0 + s 0 .**

- **set trace on/off .**: to turn on/off the flag to display (hide) execution trace.

- **show module NAME .**: to display the code of module named **NAME**.

- **quit .**: to quit Maude

Note that **.** is necessary at the end of some commands.

Some basic Math notations in Maude

A Maude example

```
1 fmod NAT is                      --- declare a module
2   sort Nat .                         --- declare a sort
3   op 0 : -> Nat .                  --- declare an operator
4   op s_ : Nat -> Nat .           --- mix-fix operator
5   op _#_ : Nat Nat -> Nat .      --- assoc means associativity
6
7   vars X Y : Nat .                --- declare variables
8   eq 0 # X = X .                 --- eq means equation
9   eq s X # Y = s(X # Y) .
10 endfm                           --- end of the module
```

Equational simplification

Why can $s\ 0 + s\ 0$ be reduced to $s\ s\ 0$ with the following two equations?

1 **eq** $0 + X = X$.

2 **eq** $s\ X + Y = s\ (X + Y)$.

Definition of factorial function

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times fac(n - 1) & \text{if } n > 0 \end{cases}$$

Maude code of function `fac`

```
1 fmod FAC is
2   including NAT .
3
4   op fac : Nat -> Nat .
5
6   var N : Nat .
7   eq fac(0) = 1 .
8   eq fac(s N) = (s N) * fac(N) .
9 endfm
```

Methodology: **Definition is implementation!**

C implementation of factorial function

$$fac(n) = \begin{cases} 1 & \text{if } n = 0 \\ x \times fac(n - 1) & \text{if } n > 0 \end{cases}$$

C code of function **fac**

```
1 int fac(int x){  
2     int n=1;  
3     while(x>0){  
4         n=n*x;  
5         x--;  
6     }  
7     return n;  
8 }
```

Methodology: There are gaps between definition and implementation!

Quiz

$$fac(1000) = ?$$

Maude is more powerful!

```
1 Maude> red fac(1000) .
2 reduce in FAC : fac(1000) .
3 rewrites: 2001
4 result NzNat:
   40238726007709377354370243392300398571937486421071463254379991042993851239
5 8629020592044208486969404800479988610197196058631666872994808558901323829669944
6 5909974245040870737599188236277271887325197795059509952761208749754624970436014
7 1827809464649629105639388743788648733711918104582578364784997701247663288983595
8 5735432513185323958463075557409114262417474349347553428646576611667797396668820
9 2912073791438537195882498081268678383745597317461360853795345242215865932019280
10 9087829730843139284440328123155861103697680135730421616874760967587134831202547
11 8589320767169132448426236131412508780208000261683151027341827977704784635868170
12 1643650241536913982812648102130927612448963599287051149649754199093422215668325
13 7208082133318611681155361583654698404670897560290095053761647584772842188967964
14 6244945160765353408198901385442487984959953319101723355556602139450399736280750
15 1378376153071277619268490343526252000158885351473316117021039681759215109077880
16 1939317811419454525722386554146106289218796022383897147608850627686296714667469
17 7562911234082439208160153780889893964518263243671616762179168909779911903754031
18 2746222899880051954444142820121873617459926429565817466283029555702990243241531
19 8161721046583203678690611726015878352075151628422554026517048330422614397428693
20 3061690897968482590125458327168226458066526769958652682272807075781391858178889
```

Maude is more powerful!

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0)
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0)
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0) ==>  
(s s s s s 0) * (s s s s 0) * fac(s s s 0)
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0) ==>  
(s s s s s 0) * (s s s s 0) * fac(s s s 0) ==>  
20 * fac(s s s 0)
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0) ==>  
(s s s s s 0) * (s s s s 0) * fac(s s s 0) ==>  
20 * fac(s s s 0) ==>  
20 * (s s s 0) * fac(s s 0)
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0) ==>  
(s s s s s 0) * (s s s s 0) * fac(s s s 0) ==>  
20 * fac(s s s 0) ==>  
20 * (s s s 0) * fac(s s 0) ==>  
60 * fac(s s 0)
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0) ==>  
(s s s s s 0) * (s s s s 0) * fac(s s s 0) ==>  
20 * fac(s s s 0) ==>  
20 * (s s s 0) * fac(s s 0) ==>  
60 * fac(s s 0) ==>  
60 * (s s 0) * fac(s 0)
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0) ==>  
(s s s s s 0) * (s s s s 0) * fac(s s s 0) ==>  
20 * fac(s s s 0) ==>  
20 * (s s s 0) * fac(s s 0) ==>  
60 * fac(s s 0) ==>  
60 * (s s 0) * fac(s 0) ==>  
120 * fac(s 0)
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0) ==>  
(s s s s s 0) * (s s s s 0) * fac(s s s 0) ==>  
20 * fac(s s s 0) ==>  
20 * (s s s 0) * fac(s s 0) ==>  
60 * fac(s s 0) ==>  
60 * (s s 0) * fac(s 0) ==>  
120 * fac(s 0) ==>  
120 * (s 0) * fac(0)
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0) ==>  
(s s s s s 0) * (s s s s 0) * fac(s s s 0) ==>  
20 * fac(s s s 0) ==>  
20 * (s s s 0) * fac(s s 0) ==>  
60 * fac(s s 0) ==>  
60 * (s s 0) * fac(s 0) ==>  
120 * fac(s 0) ==>  
120 * (s 0) * fac(0) ==>  
120 * fac(0)
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0) ==>  
(s s s s s 0) * (s s s s 0) * fac(s s s 0) ==>  
20 * fac(s s s 0) ==>  
20 * (s s s 0) * fac(s s 0) ==>  
60 * fac(s s 0) ==>  
60 * (s s 0) * fac(s 0) ==>  
120 * fac(s 0) ==>  
120 * (s 0) * fac(0) ==>  
120 * fac(0) ==>  
120 * 1
```

How does Maude compute?

Computation: Term rewriting!

```
eq fac(0) = 1 .  
eq fac(s N) = (s N) * fac(N) .
```

Computation process:

```
fac(s s s s s 0) ==>  
(s s s s s 0) * fac(s s s s 0) ==>  
(s s s s s 0) * (s s s s 0) * fac(s s s 0) ==>  
20 * fac(s s s 0) ==>  
20 * (s s s 0) * fac(s s 0) ==>  
60 * fac(s s 0) ==>  
60 * (s s 0) * fac(s 0) ==>  
120 * fac(s 0) ==>  
120 * (s 0) * fac(0) ==>  
120 * fac(0) ==>  
120 * 1 ==>
```

Convention in Maude

- Module name: capital. e.g., `NAT`
- Sort name: First character is capital, e.g., `Nat`
- Variable name: single capital character, e.g., `X`
- Operator name: lower case, e.g., `fac`

Example on natural numbers

```
1 fmod PNAT is
2   sort PNat .
3   op 0 : -> PNat .
4   op s_ : PNat -> PNat .
5
6   op _+_ : PNat PNat -> PNat .
7
8   op _*_ : PNat PNat -> PNat .
9
10 vars X Y : PNat .
11
12 eq 0 + X = X .
13 eq s X + Y = s (X + Y) .
14
15 eq 0 * X = 0 .
16 eq s X * Y = (X * Y) + Y .
17 endfm
```

Example on natural numbers

What happens when you try the following examples:

```
1 red s s 0 + s s s 0 + s s s s 0 .
2 red s s 0 * s s s 0 + s s s s 0 .
```

```
1 Warning: <standard input>, line 25: ambiguous term, two parses are:
```

```
2 s s 0 + (s s s 0 + s s s s 0)
```

```
3 -versus-
```

```
4 (s s 0 + s s s 0) + s s s s 0
```

```
5
```

```
6 Arbitrarily taking the first as correct.
```

```
7 reduce in PNAT : s s 0 + (s s s 0 + s s s s 0) .
```

```
8 rewrites: 7
```

```
9 result PNat: s s s s s s s s 0
```

Coding in Maude: precedence of operators

Operators have precedences, e.g., $pri(+)>pri(*)$.

```
1 fmod PNAT is
2   sort PNat .
3   op 0 : -> PNat .
4   op s_ : PNat -> PNat .
5   op _+_ : PNat PNat -> PNat [prec 30] .
6   op _*_ : PNat PNat -> PNat [prec 28] .
7
8   vars X Y : PNat .
9   eq 0 + X = X .
10  eq s X + Y = s (X + Y) .
11
12  eq 0 * X = 0 .
13  eq s X * Y = (X * Y) + Y .
14 endfm
```

Precedence of term

- A constant's precedence is 0;
- A variable's precedence is 0;
- A term $f(t_1, \dots, t_n)$'s precedence is equal to the precedence of f .

Coding in Maude: *gather*

To clear ambiguous parsing by *gather* attribute.

Syntax: *gather(E e &)*, to assign a value i.e., *E*, *e* or *&* to each parameter of the operator.

- *E*: The parameter's precedence value must be lower than or equal to (\leq) the operator's.
- *e*: strictly less than <
- *&*: arbitrary precedence

```

1 fmod PNAT is
2   sort PNat .
3   op 0 : -> PNat .
4   op s_ : PNat -> PNat .
5   op _+_ : PNat PNat -> PNat [prec 30 gather(E e)] .
6   op _*_ : PNat PNat -> PNat [prec 28 gather(E e)] .
7
8   vars X Y : PNat .
9   eq 0 + X = X .
10  eq s X + Y = s (X + Y) .
11  eq 0 * X = 0 .
12  eq s X * Y = (X * Y) + Y .
13 endfm

```

Coding in Maude: *gather*

```
1 reduce in PNAT : s s 0 + s s s 0 + s s s s 0 .
2 rewrites: 9
3 result PNat: s s s s s s s s s 0
```

WHY?

Coding in Maude: List

Your mission: *to define a **list** of natural numbers.*

What is a **list**?

Coding in Maude: List

Your mission: *to define a **list** of natural numbers.*

What is a **list**?

Definition (List)

In computer science, a list or sequence is an **abstract data type** that represents **an ordered sequence of values**, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a **finite** sequence; the (potentially) infinite analog of a list is a stream.

Essence: *list is a finite ordered sequence.*

Coding in Maude: List-Nat

```
1 fmod LIST-NAT is
2   protecting NAT .
3
4   sort List-Nat .
5   subsort Nat < List-Nat .
6
7   op nil :           -> List-Nat [ctor] .
8   op _,_ : Nat List-Nat -> List-Nat [ctor] .
9 endfm
```

- **ctor**: constructor
- **subsort**: strict partial order relation on sort.

Coding in Maude: List-Nat

Operations on **List-Nat**: the length of list

```
1 fmod LIST-NAT is
2   protecting NAT .
3
4   sort List-Nat .
5   subsort Nat < List-Nat .
6
7   op nil :           -> List-Nat [ctor] .
8   op _,_ : Nat List-Nat -> List-Nat [ctor] .
9
10  op len : List-Nat -> Nat .
11  var LN : List-Nat .
12  var N : Nat .
13  eq len(nil) = 0 .
14  eq len(N, LN) = len(LN) + 1 .
15 endfm
16
17 red len(1,2,3) .
18 reduce in LIST-NAT : len(1,2,3) .
19 rewrites: 3
20
21 N = Nat & S = List-Nat
```

Coding in Maude: List-Nat

Operations on **List-Nat**: the length of list

```
1 fmod LIST-NAT is
2   protecting NAT .
3
4   sort List-Nat .
5   subsort Nat < List-Nat .
6
7   op nil :           -> List-Nat [ctor] .
8   op _,_ : Nat List-Nat -> List-Nat [ctor] .
9
10  op len : List-Nat -> Nat .
11  var LN : List-Nat .
12  var N : Nat .
13  eq len(nil) = 0 .
14  eq len(N, LN) = len(LN) + 1 .
15  eq len(N) = 1 .
16 endfm
17
18 red len(1,2,3) .
19 reduce in LIST-NAT : len(1,2,3) .
```

Coding in Maude: List-Nat

Operations on **List-Nat**: the length of list

```
1 fmod LIST-NAT is
2   protecting NAT .
3
4   sort List-Nat .
5   subsort Nat < List-Nat .
6
7   op nil :           -> List-Nat [ctor id: nil] .
8   op _,_ : Nat List-Nat -> List-Nat [ctor] .
9
10  op len : List-Nat -> Nat .
11  var LN : List-Nat .
12  var N : Nat .
13  eq len(nil) = 0 .
14  eq len(N, LN) = len(LN) + 1 .
15 endfm
16
17 red len(1,2,3) .
18 reduce in LIST-NAT : len(1,2,3) .
19 rewrites: 4
```

Identity

Identity is a special element e in a set A with respect to an operation e.g., \cdot such that:

$$\forall a \in A. e \cdot a = a \quad (1)$$

$$\forall a \in A. a \cdot e = a \quad (2)$$

If only equation (1) is satisfied, e is called a **left identity**

If only equation (2) is satisfied, e is called a **right identity**

Obviously, $2 = 2, \text{nil} = 2, \text{nil}, \text{nil} = \dots$

Coding in Maude: List-Nat

Operations on **List-Nat**: the length of list.

Use of identity

```
1 Maude> red len(0) .
2 reduce in LIST-NAT : len(0) .
3 ***** equation eq len(N, LN) = 1 + len(LN) .
4 N --> 0
5 LN --> nil
6 len(0)--->1 + len(nil)
7 ***** equation eq len(nil) = 0 .
8 empty substitution
9 len(nil)--->0
10 ***** equation(built-in equation for symbol _+_)
11 0 + 1--->1
12 rewrites: 3
13 result NzNat: 1
```

Your homework

- 1 to implement function **revert**, which reverts the order of the natural numbers in a given list.
e.g. **revert(3,2,7,1)=1,7,2,3**
- 2 to implement function **sorting**, which sorts the natural numbers in a given list in the ascend order
e.g. **sorting(3,2,7,1)=1,2,3,7**

Lecture 03: Functional Module

What is a functional module

```
1 fmod MODULE-NAME is
2   including/protecting/extending MODULE-NAME1
3   sort S1 S2 S3 .
4   subsort S1 < S2 .
5   op f1 f2 f3 : S1 ... -> Sn
6   var V1 V2 V3 : S1 .
7   eq f1(V1,...) = xxx .
8   ceq f2(V2,...) = yyy if ... .
9   mb V1; V2 : S1 .
10  cmb V2 ; V3 : S2 if zzz .
11 endfm
```

What is functional module used for

- Define/Formalize a class of objects
- Define/Formalize operations on objects

What is functional module used for

- Define/Formalize a class of objects
- Define/Formalize operations on objects

If you want to know the essence of something, define it in Maude!

More important:

To study properties of operations

Exercises

- Computation of square root
- Search and binary search
- Binary Search Tree

Square root

For any positive integer n , there exists a natural number r such that $r \leq \sqrt{n} < (r + 1)$.

To calculate the integeral value r of a positive integer n 's square root:

Algorithm:

- 1 $r := 0;$
- 2 If $r^2 \leq n < (r + 1)^2$ is true, return r ;
- 3 Otherwise, $r ++;$
- 4 Repeat step 2 and 3

Definition of `sqrt` in Maude

```
1 op sqrt : Int -> Nat .
2 var N : Int .
3 eq sqrt(N) = ?? .
```

Definition of `sqrt` in Maude

```

1 op sqrt : Nat -> Nat .
2 op sqrt' : Nat Nat -> Nat .
3 vars N N' : Nat .
4 eq sqrt(N) = sqrt'(N,0) .
5 eq sqrt'(N,N') = (if N' * N' <= N and N < (N' + 1) * (N' + 1)
6 then
7 N'
8 else
9 sqrt'(N,N' + 1)
10 fi) .

```

```

1 red sqrt(200000000) .
2 rewrites: 113145
3 Result: 14142 .

```

If n 's arithmetic square root is r , the number of iteration is $r + 1$.

Computation of an arithmetic square root

If n 's arithmetic square root is r , then $0 \leq r \leq n$.

For any positive integer x such that $0 \leq x \leq n$:

- If $x^2 < n$, then $x \leq r \leq n$.

Example: $n = 101$, $x = 9$, $9 \leq r \leq 101$

$0 \dots \textcolor{red}{x} \dots \textcolor{red}{r} \dots n$

- If $x^2 > n$, then $0 \leq r \leq x - 1$.

Example: $n = 101$, $x = 11$, $0 \leq r \leq 10$

$0 \dots \textcolor{red}{r} \dots \textcolor{red}{x} \dots n$

Computation of an arithmetic square root

If n 's arithmetic square root is r , and it satisfies $l \leq r \leq u$, then for any positive integer x such that $l \leq x \leq u$:

- If $x^2 < n$, then $x \leq r \leq u$.

Example: $n = 101$, $x = 9$, $l = 4$, $u = 14$, then $9 \leq r \leq 14$

$l \dots x \dots r \dots u$

- If $x^2 > n$, then $l \leq r \leq x - 1$.

Example: $n = 101$, $x = 11$, $l = 4$, $u = 14$, $4 \leq r \leq 10$

$l \dots r \dots x \dots u$

Another definition of `sqrt` in Maude (II)

```

1 op sqrt2 : Nat -> Nat .
2 op sqrt2' : Nat Nat Nat Nat -> Nat .
3 op pivot : Nat Nat -> Nat .
4 vars L U X N : Nat .
5 eq sqrt2(N) = sqrt2'(0,N,pivot(0,N),N) .
6 eq sqrt2'(L,U,X,N) = (if L == U then L else
7   (if X * X > N
8     then sqrt2'(L, sd(X,1), pivot(L,sd(X,1)),N)
9     else sqrt2'(X,U,pivot(X,U),N)
10    fi)
11  fi) .
12 eq pivot(L,U) =
13   (if 2 divides sd(U,L) == 0 then sd(U,L) quo 2 + L else
14     (sd(U,L) + 1) quo 2 + L fi) .

```

```

1 red sqrt2(200000000) .
2 rewrites: 373
3 Result: 14142 .

```

Binary search and linear search

- Binary search

After an iteration, the searching space is decreased by the half

- Linear search

After an iteration, the search space is decreased by 1

Binary Search Tree

Binary Search Tree

Definition of BST in Maude

```
1 fmod BINARY-TREE is
2   including NAT .
3
4   sorts BTree Leaf .
5   subsort Leaf < BTree .
6   op leaf : -> Leaf .
7   op ((_,_,_)) : BTree Nat BTree -> BTree .
8
9 endfm
```

Example:

```
1 (leaf,3,leaf)
2 ((leaf,3,leaf),2,leaf)
3 ((leaf,3,leaf),2,(leaf,5,leaf))
```

Definition of BST

Definition (BST)

A binary tree t is a binary search tree if and only if one of the following two conditions is satisfied:

- 1 t is an empty tree;
- 2 Both the left sub-tree and right sub-tree of t are binary search tree, and
 - Any child in the left sub-tree is less than the root
 - Any child in the right sub-tree is greater than the root

Basic operations on BST: to check if a tree is BST

```
1 fmod BINARY-TREE is
2   including NAT .
3
4   sorts BTree Leaf .
5   subsort Leaf < BTree .
6   op leaf : -> Leaf .
7   op ((_,_,_)) : BTree Nat BTree -> BTree .
8
9   op isBSTree : BTree -> Bool .
10  eq isBSTree(leaf) = true .
11  vars LT RT : BTree .
12  var N : Nat .
13  eq isSBTree((LT,N,RT)) = _____ .
14 endfm
```

Basic operations on BST: to check if a tree is BST

```
1 fmod BINARY-TREE is
2   including NAT .
3
4   sorts BTree Leaf .
5   subsort Leaf < BTree .
6   op leaf : -> Leaf .
7   op ((_,_,_)) : BTree Nat BTree -> BTree .
8
9   op isBSTree : BTree -> Bool .
10  eq isBSTree(leaf) = true .
11  vars LT RT : BTree .
12  var N : Nat .
13  ceq isBSTree((LT,N,RT)) = isBSTree(LT) and isBSTree(RT) and
14    right-most(LT) < N and left-most(RT) > N
15  if LT =/= leaf /\ RT =/= leaf .
16
17
18
19
20 endfm
```

Basic operations on BST: to check if a tree is BST

```
1 fmod BINARY-TREE is
2   including NAT .
3
4   sorts BTree Leaf .
5   subsort Leaf < BTree .
6   op leaf : -> Leaf .
7   op ((_,_,_)) : BTree Nat BTree -> BTree .
8
9   op isBSTree : BTree -> Bool .
10  eq isBSTree(leaf) = true .
11  vars LT RT : BTree .
12  var N : Nat .
13  ceq isBSTree((LT,N,RT)) = isBSTree(LT) and isBSTree(RT) and
14    right-most(LT) < N and left-most(RT) > N
15  if LT =/= leaf /\ RT =/= leaf .
16  ceq isBSTree((leaf,N,RT)) = isBSTree(RT) and left-most(RT) > N
17  if RT =/= leaf .
18  ceq isBSTree((LT,N,leaf)) = isBSTree(LT) and right-most(RT) < N
19  if LT =/= leaf .
20  eq isBSTree((leaf,N,leaf)) = true .
```

Basic operations on BST: to check if a tree is BST

```
1 fmod BINARY-TREE is
2   including NAT .
3
4   sorts BTree Leaf .
5   subsort Leaf < BTree .
6   op leaf : -> Leaf .
7   op ((_,_,_)) : BTree Nat BTree -> BTree .
8
9
10  op right-most : BTree -> Nat .
11  op left-most : BTree -> Nat .
12
13  ceq right-most((LT,N,RT)) = right-most(RT) if RT =/= leaf .
14  eq right-most((LT,N,leaf)) = N .
15  ceq left-most((LT,N,RT)) = left-most(LT) if LT =/= leaf .
16  eq right-most((leaf,N,RT)) = N .
17
18 endfm
```

Basic operations on BST: to insert a node

```
1 fmod BINARY-TREE is
2   including NAT .
3
4   sorts BTree Leaf .
5   subsort Leaf < BTree .
6   op leaf : -> Leaf .
7   op ((_,_,_)) : BTree Nat BTree -> BTree .
8
9   op insert : BTree Nat -> BTree .
10
11
12 endfm
```

Basic operations on BST: to insert a node

```
1 fmod BINARY-TREE is
2   including NAT .
3
4   sorts BTree Leaf .
5   subsort Leaf < BTree .
6   op leaf : -> Leaf .
7   op ((_,_,_)) : BTree Nat BTree -> BTree .
8
9   op insert : BTree Nat -> BTree .
10  eq insert(leaf,N) = (leaf,N,leaf) .
11  vars LT RT : BTree . vars N N' : Nat .
12  eq insert((LT,N',RT),N) =
13    (if N' > N then (insert(LT,N),N',RT)
14     else
15       (if N' < N then (LT,N',insert(RT,N))
16        else (LT,N',RT) fi)
17     fi) .
18 endfm
```

```
1 red insert(((leaf,1,leaf),2,(leaf,8,leaf)),6) .
```

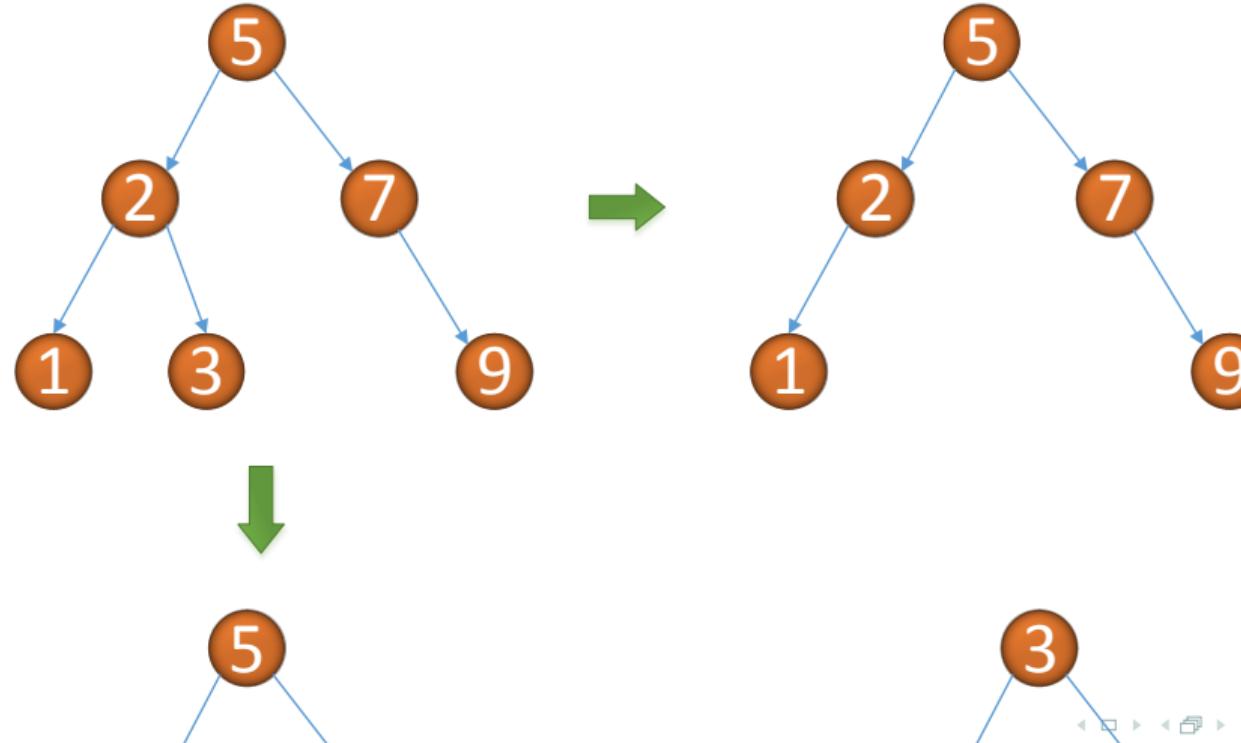
Basic operations on BST: to search a node

```
1 fmod BINARY-TREE is
2   including NAT .
3
4   sorts BTree Leaf .
5   subsort Leaf < BTree .
6   op leaf : -> Leaf .
7   op ((_,_,_)) : BTree Nat BTree -> BTree .
8
9   op search : BTree Nat -> Bool .
10  eq search(leaf,N) = false .
11  vars LT RT : BTree . vars N N' : Nat .
12  ceq search((LT,N',RT),N) = (if N' > N then search(LT,N)
13    else
14      (if N' < N then search(RT,N)
15        else true fi)
16    fi)
17  if isBSTree((LT,N',RT)) .
18 endfm
```

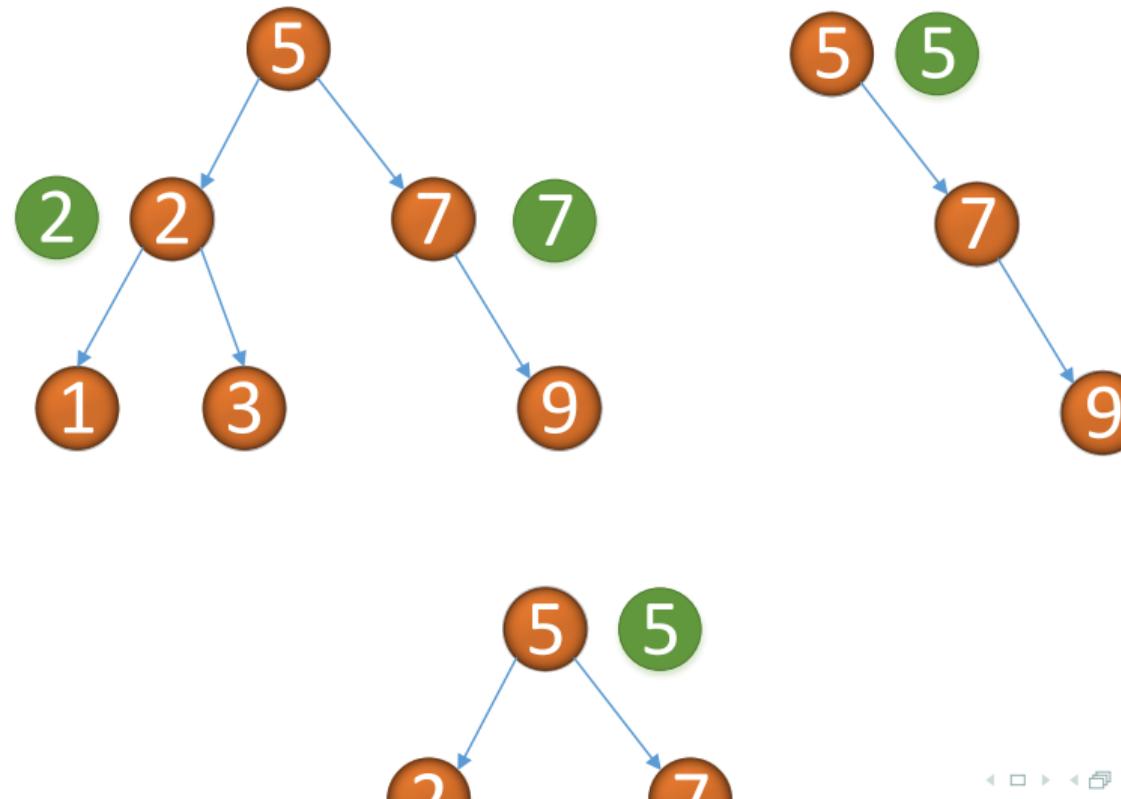
Basic operations on BST: to search a node

```
1 fmod BINARY-TREE is
2   including NAT .
3
4   sorts BTREE Leaf .
5   subsort Leaf < BTREE .
6   op leaf : -> Leaf .
7   op ((_,_,_)) : BTREE Nat BTREE -> BTREE .
8
9   op remove : BTREE Nat -> BTREE .
10  eq remove(leaf,N) = leaf .
11  eq remove((LT,N',RT),N) = _____ .
12 endfm
```

Basic operations on BST: to delete a node



Basic operations on BST: to delete a node



Basic operations on BST: to delete a node

```

1 fmod BINARY-TREE is
2   including NAT .
3   sorts BTree Leaf .
4   subsort Leaf < BTree .
5   op leaf : -> Leaf .
6   op ((_,_,_)) : BTree Nat BTree -> BTree .

7
8   op remove : BTree Nat -> BTree .
9   eq remove(leaf,N) = leaf .
10  ceq remove((LT,N,RT),N') = (remove(LT,N'),N,RT) if N > N' /\ isBSTree((LT,N,RT)) .
11  ceq remove((LT,N,RT),N') = (LT,N,remove(RT,N')) if N < N' /\ isBSTree((LT,N,RT)) .
12  ceq remove((LT,N,RT),N) =
13    (if LT == leaf then RT
14     else (rm-right-most(LT),right-most(LT),RT) fi)
15  if isBSTree((LT,N,RT)) .
16 endfm

```

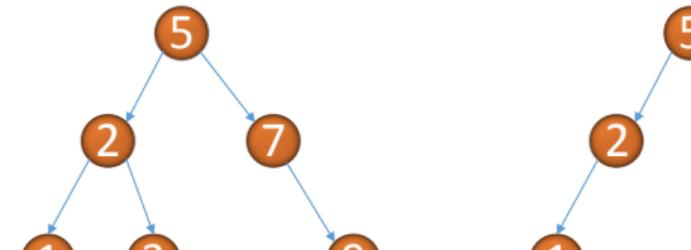
```

1 red remove(((leaf,1,leaf),3,leaf),5,(leaf,7,(leaf,9,leaf))),5) .
2 result BTree: ((leaf,1,leaf),3,(leaf,7,(leaf,9,leaf)))

```

Basic operations on BST: to delete a node

```
1 fmod BINARY-TREE is
2   including NAT .
3
4   sorts BTree Leaf .
5   subsort Leaf < BTree .
6   op leaf : -> Leaf .
7   op ((_,_,_)) : BTree Nat BTree -> BTree .
8
9   op rm-right-most : BTree -> BTree .
10  eq rm-right-most(leaf) = leaf .
11  ceq rm-right-most((LT,N,RT)) = rm-right-most(RT) if RT =/= leaf .
12  eq rm-right-most((LT,N,leaf)) = LT .
13 endfm
```



Lecture 04: System Module

描述过河问题：判定是否安全

```
1 mod SYSTEM-MODULE is
2   sorts AA BB .
3   ops . . .
4   op
5
6   eq X + Y = . . .
7   mb
8   cmb
9
10  rl X ; Y => . . .
11  crl X ; Y => . . . if X > ??? .
12 endm
```

过河问题



- 船夫一次只能带一样过河
- 船夫不在时，羊吃白菜，狼吃羊

描述过河问题：判定是否安全

```
1 mod RIVER-CROSSING is
2   sorts Side Group .
3   ops left right : -> Side [ctor] .
4   op change : Side -> Side .
5   eq change(left) = right .
6   eq change(right) = left .
7
8   ops s w l c : Side -> Group [ctor] .
9   op __ : Group Group -> Group [ctor assoc comm] .
10
11 vars S S1 S2 S3 : Side .
12
13 op isSafe : Group -> Bool .
14 eq isSafe(s(S) G:Group) = true .
15
16
17
18
19 ...
20 endm
```

描述过河问题：判定是否安全

```
1 mod RIVER-CROSSING is
2   sorts Side Group .
3   ops left right : -> Side [ctor] .
4   op change : Side -> Side .
5   eq change(left) = right .
6   eq change(right) = left .
7
8   ops s w l c : Side -> Group [ctor] .
9   op __ : Group Group -> Group [ctor assoc comm] .
10
11 vars S S1 S2 S3 : Side .
12
13 op isSafe : Group -> Bool .
14 eq isSafe(s(S) G:Group) = true .
15 eq isSafe(w(S1) l(S2) c(S3)) = (S1 =/= S2) and (S2 =/= S3) .
```

描述过河问题：判定是否安全

```
1 mod RIVER-CROSSING is
2   sorts Side Group .
3   ops left right : -> Side [ctor] .
4   op change : Side -> Side .
5   eq change(left) = right .
6   eq change(right) = left .
7
8   ops s w l c : Side -> Group [ctor] .
9   op __ : Group Group -> Group [ctor assoc comm] .
10
11 vars S S1 S2 S3 : Side .
12
13 op isSafe : Group -> Bool .
14 eq isSafe(s(S) G:Group) = true .
15 eq isSafe(w(S1) l(S2) c(S3)) = (S1 =/= S2) and (S2 =/= S3) .
16 eq isSafe(w(S1) l(S2)) = (S1 =/= S2) .
17
18
19
20
21
```

描述过河问题：判定是否安全

```
1 mod RIVER-CROSSING is
2   sorts Side Group .
3   ops left right : -> Side [ctor] .
4   op change : Side -> Side .
5   eq change(left) = right .
6   eq change(right) = left .
7
8   ops s w l c : Side -> Group [ctor] .
9   op __ : Group Group -> Group [ctor assoc comm] .
10
11 vars S S1 S2 S3 : Side .
12
13 op isSafe : Group -> Bool .
14 eq isSafe(s(S) G:Group) = true .
15 eq isSafe(w(S1) l(S2) c(S3)) = (S1 =/= S2) and (S2 =/= S3) .
16 eq isSafe(w(S1) l(S2)) = (S1 =/= S2) .
17 eq isSafe(l(S2) c(S3)) = (S2 =/= S3) .
```

描述过河问题：判定是否安全

```
1 mod RIVER-CROSSING is
2   sorts Side Group .
3   ops left right : -> Side [ctor] .
4   op change : Side -> Side .
5   eq change(left) = right .
6   eq change(right) = left .
7
8   ops s w l c : Side -> Group [ctor] .
9   op __ : Group Group -> Group [ctor assoc comm] .
10
11 vars S S1 S2 S3 : Side .
12
13 op isSafe : Group -> Bool .
14 eq isSafe(s(S) G:Group) = true .
15 eq isSafe(w(S1) l(S2) c(S3)) = (S1 =/= S2) and (S2 =/= S3) .
16 eq isSafe(w(S1) l(S2)) = (S1 =/= S2) .
17 eq isSafe(l(S2) c(S3)) = (S2 =/= S3) .
18 eq isSafe(w(S1) c(S2)) = true .
19
20
21
```

描述过河问题: 过河规则

如果安全, 船夫可以在两岸之间来回游动。

```
1 ...
2 crl [shepherd] : s(S) w(S1) l(S2) c(S3) =>
3   s(change(S)) w(S1) l(S2) c(S3) if isSafe(w(S1) l(S2) c(S3)) .
4
5
6
7
8
9
10
11
12 endm
```

描述过河问题: 过河规则

如果安全，船夫可以带着狼到对岸。

```
1 ...
2 crl [shepherd] : s(S) w(S1) l(S2) c(S3) =>
3   s(change(S)) w(S1) l(S2) c(S3) if isSafe(w(S1) l(S2) c(S3)) .
4
5 crl [wolf] : s(S) w(S) l(S1) c(S2) =>
6   s(change(S)) w(change(S)) l(S1) c(S2) if isSafe(l(S1) c(S2)) .
7
8
9
10
11
12 endm
```

描述过河问题: 过河规则

船夫可以无条件地带着羊到对岸。

```
1 ...
2 crl [shepherd] : s(S) w(S1) l(S2) c(S3) =>
3   s(change(S)) w(S1) l(S2) c(S3) if isSafe(w(S1) l(S2) c(S3)) .
4
5 crl [wolf] : s(S) w(S) l(S1) c(S2) =>
6   s(change(S)) w(change(S)) l(S1) c(S2) if isSafe(l(S1) c(S2)) .
7
8 rl [lamb] : s(S) l(S) => s(change(S)) l(change(S)) .
9
10
11
12 endm
```

描述过河问题: 过河规则

如果安全，船夫可以带着白菜到对岸。

```
1 ...
2 crl [shepherd] : s(S) w(S1) l(S2) c(S3) =>
3   s(change(S)) w(S1) l(S2) c(S3) if isSafe(w(S1) l(S2) c(S3)) .
4
5 crl [wolf] : s(S) w(S) l(S1) c(S2) =>
6   s(change(S)) w(change(S)) l(S1) c(S2) if isSafe(l(S1) c(S2)) .
7
8 rl [lamb] : s(S) l(S) => s(change(S)) l(change(S)) .
9
10 crl [cabbage] : s(S) c(S) w(S1) l(S2) =>
11   s(change(S)) c(change(S)) w(S1) l(S2) if isSafe(w(S1) l(S2)) .
12 endm
```

搜索答案

```
1 Maude> search s(left) w(left) l(left) c(left) =>*
2   s(right) w(right) l(right) c(right) .
3 Solution 1 (state 9)
```

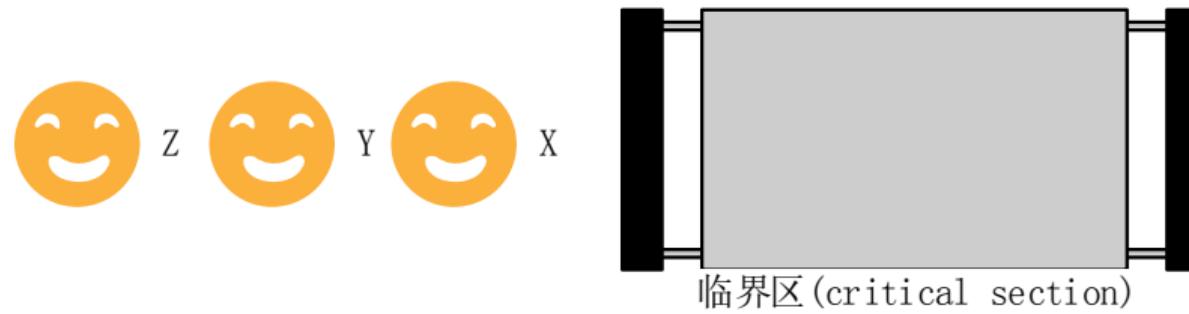
搜索答案

```
1 Maude> show path 9 .
2 state 0, Group: s(left) w(left) l(left) c(left)
3 ===[ r1 ... [label lamb] . ]===>
4 state 1, Group: s(right) w(left) l(right) c(left)
5 ===[ crl ... [label shepherd] . ]===>
6 state 2, Group: s(left) w(left) l(right) c(left)
7 ===[ crl ... [label wolf] . ]===>
8 state 3, Group: s(right) w(right) l(right) c(left)
9 ===[ r1 ... [label lamb] . ]===>
10 state 5, Group: s(left) w(right) l(left) c(left)
11 ===[ crl ... [label cabbage] . ]===>
12 state 7, Group: s(right) w(right) l(left) c(right)
13 ===[ crl ... [label shepherd] . ]===>
14 state 8, Group: s(left) w(right) l(left) c(right)
15 ===[ r1 ... [label lamb] . ]===>
16 state 9, Group: s(right) w(right) l(right) c(right)
```

操作系统中的互斥性问题 (mutual exclusion)

在任何时刻，最多只能有一个进程在临界区（critical section）。

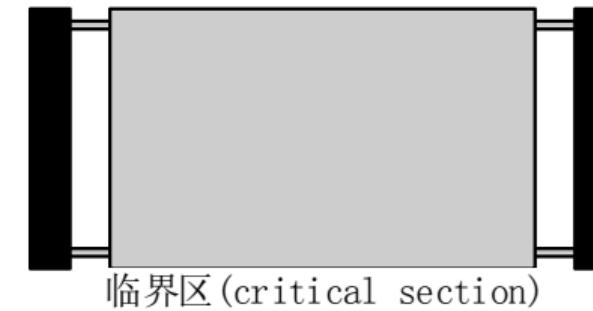
Qlock 互斥算法



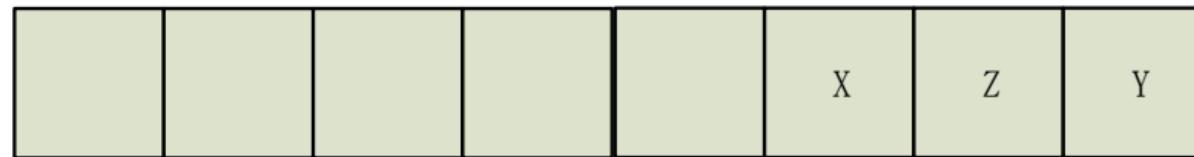
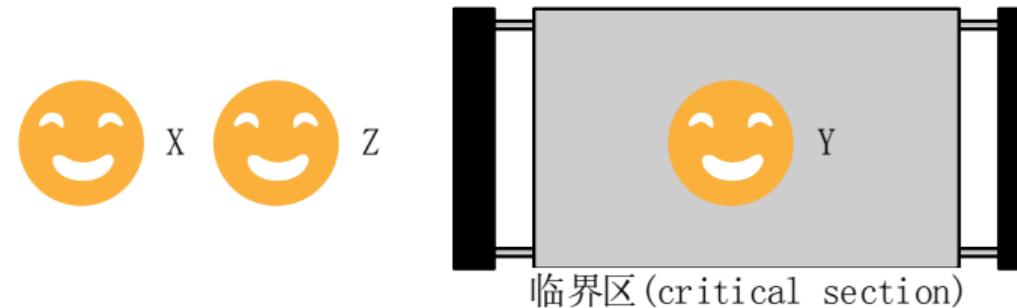
Qlock 互斥算法



Qlock 互斥算法



Qlock 互斥算法



在 Maude 中描述 Qlock

首先确定状态 (state) 所包含的信息：每个进程的位置，rs，ws, cs 和队列 queue 的值。

```
1 --- Sort Label 表示进程的位置，有三个常量分表表示 remainder section, waiting section,
2   critical section
3 fmod LABEL is
4   sort Label .
5   ops rs ws cs : -> Label [ctor] .
6 endfm
7
8 --- Pid 表示进程的标识符，这里假设只有两个进程，p1,p2
9 fmod PID is
10  sort Pid .
11  ops p1 p2 : -> Pid [ctor] .
12 endfm
```

在 Maude 中描述 Qlock

```
1 --- 定义队列及其上面的操作, enq, deq
2 fmod QUEUE is
3   pr PID .
4   sort Queue .
5   op empty : -> Queue [ctor] .
6   op _|_ : Pid Queue -> Queue [ctor] .
7   op enq : Queue Pid -> Queue .
8   op deq : Queue -> Queue .
9   var Q : Queue .
10  vars X Y : Pid .
11  eq enq(empty,X) = X | empty .
12  eq enq(Y | Q,X) = Y | enq(Q,X) .
13  eq deq(empty) = empty .
14  eq deq(X | Q) = Q .
15 endfm
```

在 Maude 中描述 Qlock

```

1 fmod QLOCK is --- 定义 Qlock 的状态, 初始状态, 以及状态转移关系
2   pr LABEL . pr QUEUE .
3   sort OCom Config . --- Config 表示状态, OCom 表示状态中的一个单元
4   subsort OCom < Config .
5   op __ : Config Config -> Config [ctor assoc comm] .
6   op pc[_]:_ : Pid Label -> OCom [ctor] .
7   op queue:_ : Queue -> OCom [ctor] .
8   op ic : -> Config .
9   eq ic = (pc[p1]: rs) (pc[p2]: rs) (queue: empty) .
10
11  var I : Pid . var Q : Queue .
12  --- 进程 I 希望进入临界区
13  rl [want] : (pc[I]: rs) (queue: Q) => (pc[I]: ws) (queue: enq(Q, I)) .
14
15  --- 如果 I 在队列的头部, I 进入临界区
16  rl [try] : (pc[I]: ws) (queue: (I | Q)) => (pc[I]: cs) (queue: (I | Q)) .
17
18  --- 进程 I 离开临界区
19  rl [exit] : (pc[I]: cs) (queue: Q) => (pc[I]: rs) (queue: deq(Q)) .
20 endfm

```

验证 Qlock 是否满足互斥

```

1 search ic =>* (pc[I:Pid]: cs) (pc[J:Pid]: cs) C:Config .
2 No solution .

```

说明如果只有两个进程时，Qlock 满足互斥。

同样，假设有 10 个进程，可做如下修改：

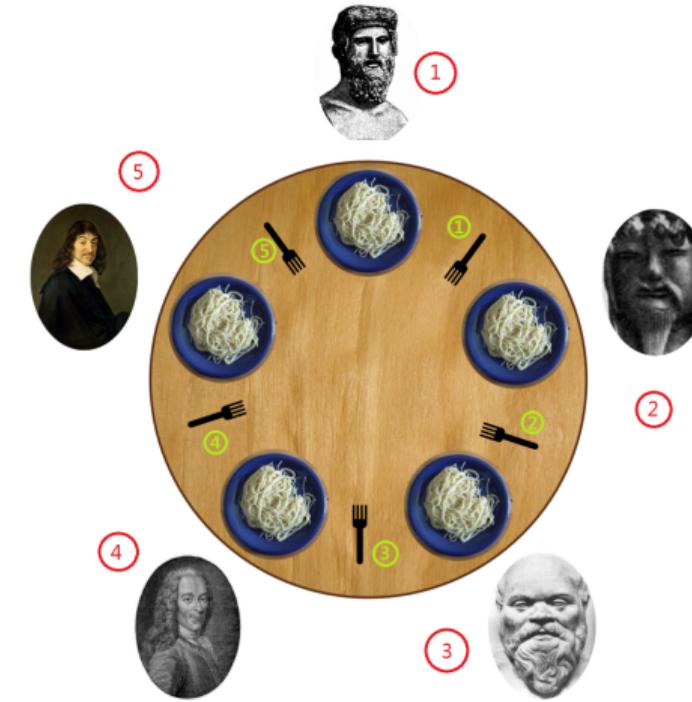
```

1 ops p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 : -> Pid . --- 定义 10 个进程的标识符
2
3 --- 定义初始状态
4 eq ic = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]: rs) (pc[p5]: rs) (pc[p6]: rs)
   (pc[p7]: rs) (pc[p8]: rs) (pc[p9]: rs) (pc[p10]: rs) (queue: empty) .
5
6 --- 搜索，这里只需要找到任意两个进程同时在临界区的情况即可
7 search ic =>* (pc[I:Pid]: cs) (pc[J:Pid]: cs) C:Config .

```

哲学家就餐问题

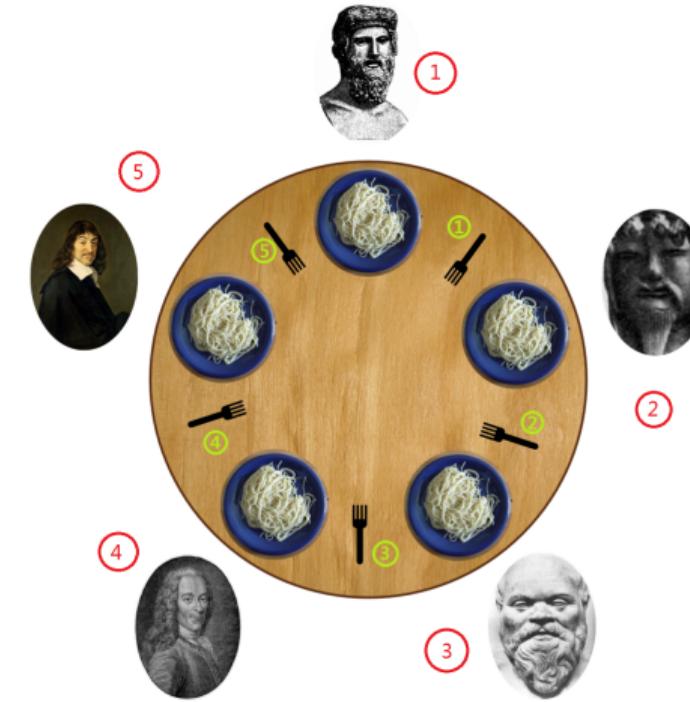
有 5 位哲学家就餐，哲学家偶尔思考，饿了就拿叉子就餐。由于条件限制，只有五个叉子，每人面前一盘拉面，要求每个人用两只叉子才可以吃面，每个哲学家只能拿他左边和右边的叉子。



哲学家就餐问题

有 5 位哲学家就餐，哲学家偶尔思考，饿了就拿叉子就餐。由于条件限制，只有五个叉子，每人面前一盘拉面，要求每个人用两只叉子才可以吃面，每个哲学家只能拿他左边和右边的叉子。

有没有可能出现都无法吃面的情况？



在 Maude 中描述哲学家问题

描述哲学家

```
1 fmod PHILOSOPHER is
2   protecting NAT .
3   sort Phi .
4   sort Status . --- 哲学家的状态
5
6   op phi : Nat -> Phi .
7   --- 如: phi(1)表示第一个哲学家。。
8
9   ops thinking eating hungry : -> Status .
10  --- thinking 哲学家在思考
11  --- eating 哲学家在就餐
12  --- hungry 哲学家饿了
13 endfm
```

在 Maude 中描述哲学家问题

描述叉子

```
1  
2 fmod FORK is  
3   protecting NAT .  
4   sorts Fork Forks .  
5  
6   --- Fork 表示叉子, Forks 表示叉子的集合  
7  
8   subsort Fork < Forks .  
9   --- 一个叉子也可以看做是一个集合, 所以 Fork 是 Forks 的 subsort  
10  
11  op empty : -> Forks .  
12  op __ : Forks Forks -> Forks [comm assoc id: empty] .  
13  
14  op fork : Nat -> Fork .  
15  
16 endfm
```

在 Maude 中描述哲学家问题

描述就餐问题

```
1 mod DINING is
2   protecting PHILOSOPHER .    --- 引用已经定义好的两个 module
3   protecting FORK .
4
5   sort State .               --- 声明一个类型表示状态
6   op p[_]:_,_ : Phi Status Nat -> State .  --- 定义状态的构造函数
7   op forks:_ : Forks -> State .  --- 定义状态的构造函数
8
9   op __ : State State -> State [assoc comm] .
10      --- 多个状态在一起组成一个新的状态
11
12
13
14
15
16
17
18
```

在 Maude 中描述哲学家问题

描述就餐问题

```
1 mod DINING is
2   protecting PHILOSOPHER .    --- 引用已经定义好的两个 module
3   protecting FORK .
4
5   sort State .              --- 声明一个类型表示状态
6   op p[_]:_,_ : Phi Status Nat -> State .    --- 定义状态的构造函数
7   op forks:_ : Forks -> State .    --- 定义状态的构造函数
8
9   op __ : State State -> State [assoc comm] .
10      --- 多个状态在一起组成一个新的状态
11   op init : -> State .    --- 声明一个状态常量 constant
12
13 eq init = ( (p[phi(1)]: thinking, 0)
14     (p[phi(2)]: thinking, 0)
15     (p[phi(3)]: thinking, 0)
16     (p[phi(4)]: thinking, 0)
17     (p[phi(5)]: thinking, 0))
18     (forks: (fork(1) fork(2) fork(3) fork(4) fork(5))) .
```

在 Maude 中描述哲学家问题

描述就餐问题

```
1 vars N N1 F : Nat .
2 var FS : Forks .
3
4 --- 如果一个哲学家正在思考，那么他进入饥饿状态
5 r1 [hungry] : (p[phi(N)]: thinking, F) => (p[phi(N)]: hungry, F) .
6
7
8
9
10
11
12
13
14
15
16
17
18
19 .
```

在 Maude 中描述哲学家问题

描述就餐问题

```
1 vars N N1 F : Nat .
2 var FS : Forks .
3
4 --- 如果一个哲学家正在思考，那么他进入饥饿状态
5 rl [hungry] : (p[phi(N)]: thinking, F) => (p[phi(N)]: hungry, F) .
6
7 --- 如果一个哲学家饿了，他要拿起叉子、
8 --- fork(N1) 表示第 N1 个叉子，如果他可以被第 N 个哲学家使用，则被拿走
9 --- 被拿走后，第 N 个哲学家的叉子个数加 1
10 crl [getFork] : (p[phi(N)]: hungry, F) (forks: (fork(N1) FS)) =>
11           (p[phi(N)]: hungry, F + 1) (forks: FS)
12   if canUse(N,N1) .
13
14
15
16
17
18
19 .
```

在 Maude 中描述哲学家问题

描述就餐问题

```

1  vars N N1 F : Nat .
2  var FS : Forks .

3
4  --- 如果一个哲学家正在思考，那么他进入饥饿状态
5  r1 [hungry] : (p[phi(N)]: thinking, F) => (p[phi(N)]: hungry, F) .

6
7  --- 如果一个哲学家饿了，他要拿起叉子、
8  --- fork(N1) 表示第 N1 个叉子，如果他可以被第 N 个哲学家使用，则被拿走
9  --- 被拿走后，第 N 个哲学家的叉子个数加 1
10 cr1 [getFork] : (p[phi(N)]: hungry, F) (forks: (fork(N1) FS)) =>
11           (p[phi(N)]: hungry, F + 1) (forks: FS)
12   if canUse(N,N1) .

13
14 op canUse : Nat Nat -> Bool .
15   --- N : 第 N 个哲学家, N1: 第 N1 个叉子,
16   --- 第 N 个哲学家可以用第 N 个和第 N-1 个叉子。
17   --- 第一个哲学家可以用第一和五个叉子
18 eq canUse(N,N1) = (N == N1) or (N == N1 + 1) or
19           (N == 1 and N1 == 5) .

```

在 Maude 中描述哲学家问题

描述就餐问题

```

1  vars N N1 F : Nat .
2  var FS : Forks .

3
4  --- 如果一个哲学家正在思考，那么他进入饥饿状态
5  r1 [hungry] : (p[phi(N)]: thinking, F) => (p[phi(N)]: hungry, F) .

6
7  --- 如果一个哲学家饿了，他要拿起叉子、
8  --- fork(N1) 表示第 N1 个叉子，如果他可以被第 N 个哲学家使用，则被拿走
9  --- 被拿走后，第 N 个哲学家的叉子个数加 1
10 cr1 [getFork] : (p[phi(N)]: hungry, F) (forks: (fork(N1) FS)) =>
11           (p[phi(N)]: hungry, F + 1) (forks: FS)
12   if canUse(N,N1) .

13
14 op canUse : Nat Nat -> Bool .
15   --- N : 第 N 个哲学家, N1: 第 N1 个叉子,
16   --- 第 N 个哲学家可以用第 N 个和第 N-1 个叉子。
17   --- 第一个哲学家可以用第一和五个叉子
18 eq canUse(N,N1) = (N == N1) or (N == N1 + 1) or
19           (N == 1 and N1 == 5) .

```

在 Maude 中描述哲学家问题

描述就餐问题

```
1   --- 如果哲学家饿了，并且有两个叉子，则开始就餐
2   r1 [eat] : (p[phi(N)]: hungry, 2) => (p[phi(N)]: eating, 2) .
3
4   --- 就餐结束后，哲学家进入思考状态，并放下两个叉子
5   --- 如果是第一个哲学家，他放下的叉子为第 1 个和第五个
6   --- 如果是第 N(N ≠ 1) 个哲学家，他放下的叉子为第 N 个和第 N - 1 个
7
8   r1 [putFork] : (p[phi(N)]: eating, 2) (forks: FS) =>
9       (p[phi(N)]: thinking, 0)
10      (forks: (FS fork(N)
11          fork(if N == 1 then 5 else sd(N, 1)))).
```

12 **endm**

利用 search 命令查看是否有死锁状态

search 死锁状态

```
1 --- 如何定义死锁 ???  
2 search init ????  
3  
4  
5  
6  
7  
8  
9  
10  
11 .
```

利用 search 命令查看是否有死锁状态

search 死锁状态

```
1 --- =>! 表示搜索一个结果使得这个结果不能被任何规则重写
2 search init =>! S:State
3
4
5
6
7
8
9
10
11 .
```

利用 search 命令查看是否有死锁状态

search 死锁状态

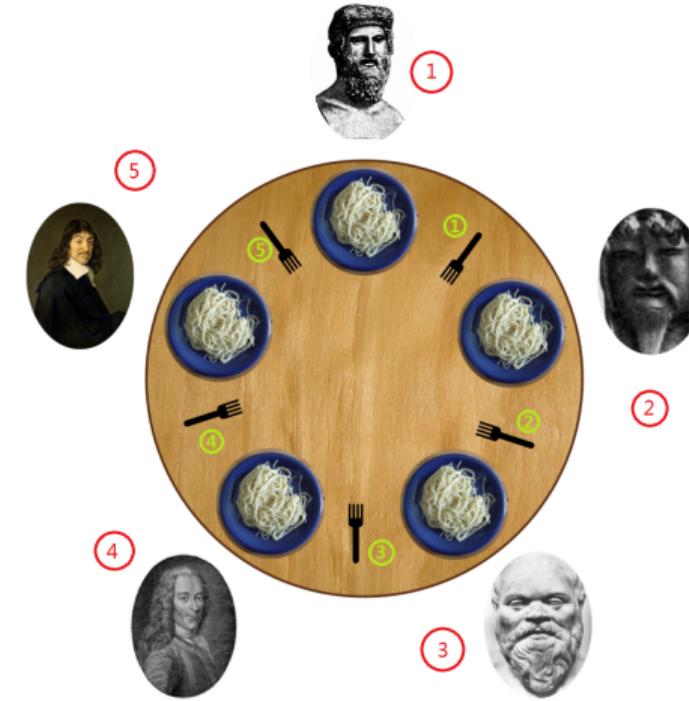
```
1 --- =>! 表示搜索一个结果使得这个结果不能被任何规则重写
2 search init =>! S:State
3
4 Solution 1 (state 1964)
5 states: 2110 rewrites: 207041
6 S:State --> (forks: empty)
7     (p[phi(1)]: hungry, 1)
8     (p[phi(2)]: hungry, 1)
9     (p[phi(3)]: hungry, 1)
10    (p[phi(4)]: hungry, 1)
11    (p[phi(5)]: hungry, 1)
```

哲学家就餐问题：解决方法一

有 5 位哲学家就餐，哲学家偶尔思考，饿了就拿叉子就餐。由于条件限制，只有五个叉子，每人面前一盘拉面，要求每个人用两只叉子才可以吃面，每个哲学家只能拿他左边和右边的叉子。

第奇数号哲学家要先拿他左边的叉子，偶数号哲学家先拿他右边的叉子

有没有可能出现都无法吃面的情况？



修改哲学家取叉子的规则

描述就餐问题

```
1 --- 如果一个哲学家饿了，他要拿起叉子、  
2 --- 如果 N 是奇数，且 N 号哲学家手上没有叉子  
3 --- 他只能拿他左边的叉子，即第 N 个叉子, rem 表示余数  
4 crl [getFork] : (p[phi(N)]: hungry, 0) (forks: (fork(N) FS)) =>  
5           (p[phi(N)]: hungry, 1) (forks: FS)  
6   if (N rem 2) == 1 .  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19 .
```

修改哲学家取叉子的规则

描述就餐问题

```

1   --- 如果一个哲学家饿了，他要拿起叉子、
2   --- 如果 N 是奇数，且 N 号哲学家手上没有叉子
3   --- 他只能拿他左边的叉子，即第 N 个叉子, rem 表示余数
4   crl [getFork] : (p[phi(N)]: hungry, 0) (forks: (fork(N) FS)) =>
5       (p[phi(N)]: hungry, 1) (forks: FS)
6   if (N rem 2) == 1 .
7
8   --- 如果一个哲学家饿了，他要拿起叉子、
9   --- 如果 N 是偶数，且 N 号哲学家手上没有叉子
10  --- 他只能拿他右边的叉子，即第 N 个叉子
11  crl [getFork] : (p[phi(N)]: hungry, 0) (forks: (fork(N') FS)) =>
12      (p[phi(N)]: hungry, 1) (forks: FS)
13  if (N rem 2) == 0 /\ N' == sd(N,1) .
14
15
16
17
18
19 .

```

修改哲学家取叉子的规则

描述就餐问题

```

1   --- 如果一个哲学家饿了，他要拿起叉子、
2   --- 如果 N 是奇数，且 N 号哲学家手上没有叉子
3   --- 他只能拿他左边的叉子，即第 N 个叉子, rem 表示余数
4   crl [getFork] : (p[phi(N)]: hungry, 0) (forks: (fork(N) FS)) =>
5       (p[phi(N)]: hungry, 1) (forks: FS)
6   if (N rem 2) == 1 .
7
8   --- 如果一个哲学家饿了，他要拿起叉子、
9   --- 如果 N 是偶数，且 N 号哲学家手上没有叉子
10  --- 他只能拿他右边的叉子，即第 N 个叉子
11  crl [getFork] : (p[phi(N)]: hungry, 0) (forks: (fork(N') FS)) =>
12      (p[phi(N)]: hungry, 1) (forks: FS)
13  if (N rem 2) == 0 /\ N' == sd(N,1) .
14
15  --- 如果一个哲学家已经拿了叉子，则他可以拿另一个叉子
16  --- 如果那个叉子可以被他拿的话
17  crl [getFork] : (p[phi(N)]: hungry, F) (forks: (fork(N') FS)) =>
18      (p[phi(N)]: hungry, F + 1) (forks: FS)
19  if F /= 0 /\ canUse(N,N') .

```

利用 search 命令查看是否有死锁状态

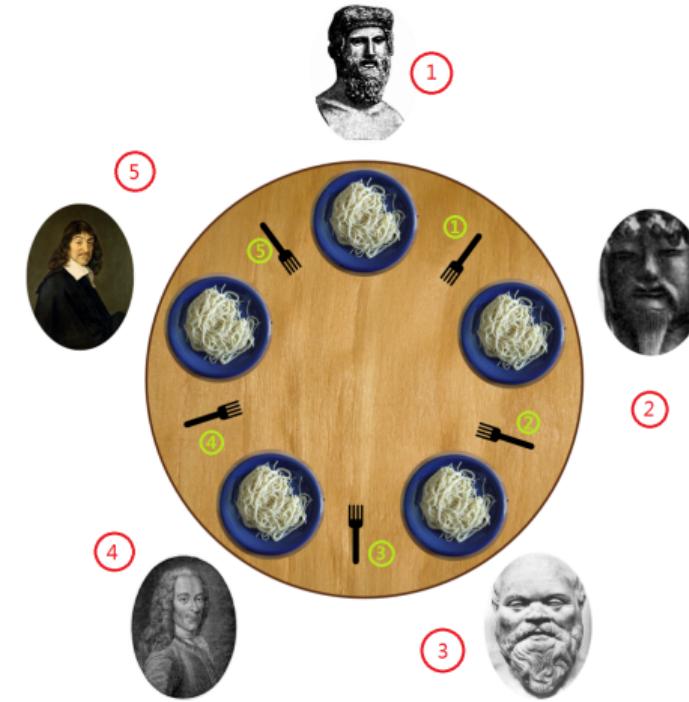
search 死锁状态

```
1 --- =>! 表示搜索一个结果使得这个结果不能被任何规则重写
2 search init =>! S:State
3
4 No Solution.
5 States: 912 ...
```

方法一的局限性

如果哲学家 5 和哲学家 3 在吃，并且 1 和 2 处于饥饿状态，那 2 可以拿叉子 1, 5 吃完后，哲学家 1 拿叉子 5，现在只能等哲学家 3 吃完后，哲学家 2 拿到叉子 2，等哲学家 2 吃完后，哲学家 1 得到叉子 1 才可以吃上面条。

然而哲学家 1 本可以拿叉子 1，等哲学家 5 吃完后就可以吃。因此对哲学家 1 不公平。

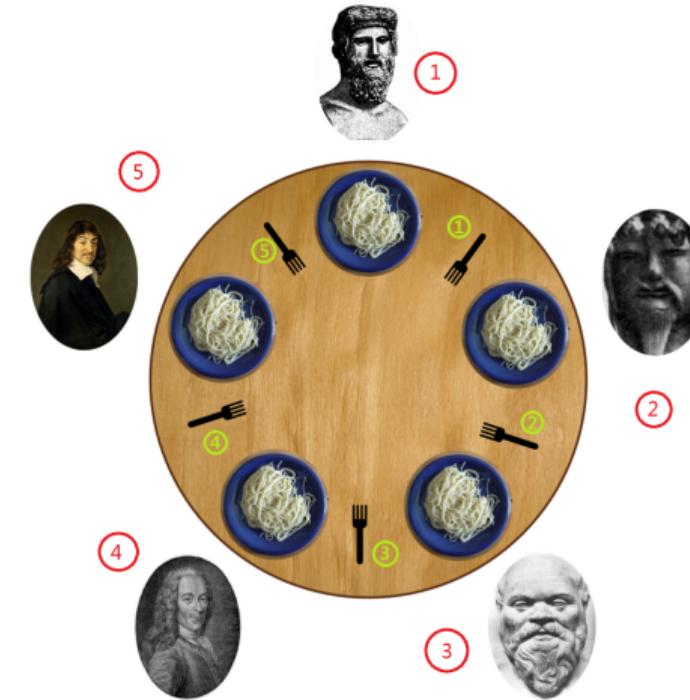


哲学家就餐问题：解决方法二

有 5 位哲学家就餐，哲学家偶尔思考，饿了就拿叉子就餐。由于条件限制，只有五个叉子，每人面前一盘拉面，要求每个人用两只叉子才可以吃面，每个哲学家只能拿他左边和右边的叉子。

每个哲学家先拿编号较小的叉子，再拿编号较高的

有没有可能出现都无法吃面的情况？



修改哲学家取叉子的规则

描述就餐问题

```

1   --- 如果一个哲学家饿了，他要拿起叉子、
2   --- 第一个哲学家第一次只能拿第五个叉子
3   rl [getFork] : (p[phi(1)]: hungry, 0) (forks: (fork(5) FS)) =>
4       (p[phi(1)]: hungry, 1) (forks: FS) .
5
6   --- 如果一个哲学家饿了，他要拿起叉子、
7   --- 其他哲学家第一次只能拿和他编号相同的叉子
8   crl [getFork] : (p[phi(N)]: hungry, 0) (forks: (fork(N) FS)) =>
9       (p[phi(N)]: hungry, 1) (forks: FS)
10      if N /= 1 .
11
12     --- 如果一个哲学家已经拿了叉子，则他可以拿另一个叉子
13     --- 如果那个叉子可以被他拿的话
14     crl [getFork] : (p[phi(N)]: hungry, F) (forks: (fork(N') FS)) =>
15         (p[phi(N)]: hungry, F + 1) (forks: FS)
16         if F /= 0 /\ canUse(N,N') .

```

利用 search 命令查看是否有死锁状态

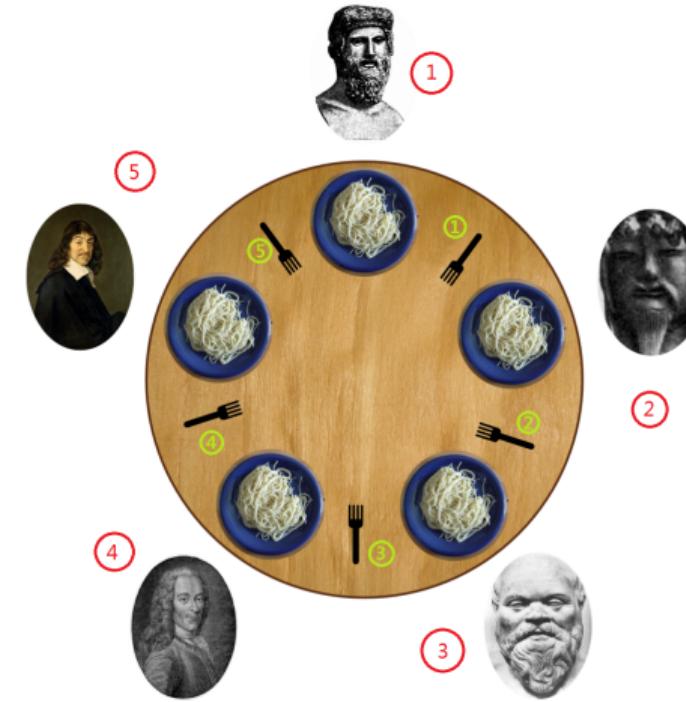
search 死锁状态

```
1 --- =>! 表示搜索一个结果使得这个结果不能被任何规则重写
2 search init =>! S:State
3
4 No Solution.
5 States: 972 ...
```

方法二同样有局限性

如果哲学家 5 和哲学家 2 在吃，并且 3 和 4 处于饥饿状态，那 4 可以拿叉子 3，哲学家 2 吃完后，哲学家 3 拿叉子 2，现在只能等哲学家 5 吃完后，哲学家 4 拿到叉子 4，等哲学家 4 吃完后，哲学家 3 得到叉子 3 才可以吃上面条。

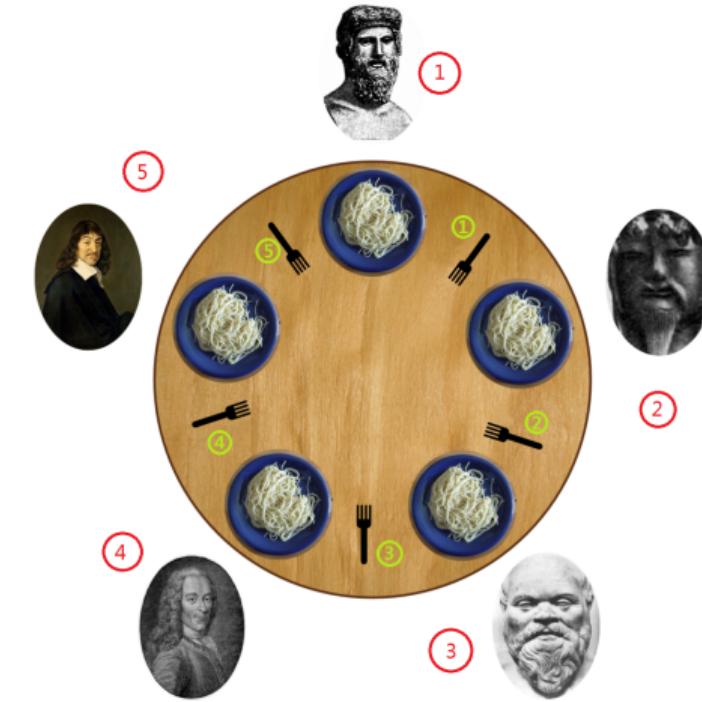
然而哲学家 3 本可以拿叉子 3，等哲学家 2 吃完后就可以吃。因此对哲学家 3 不公平。



哲学家就餐问题：解决方法三

Chandy/Misra 解法，1984 年

- 1 相邻哲学家之间允许交流
- 2 若两个哲学家竞争同一个叉子，把叉子给编号较小的人
- 3 筷子有两个状态，干净的和脏的。
初始状态下，所有的筷子都是脏的。
- 4 哲学家要吃饭时必须拿起两只叉子，当缺少某只时，他相邻居发送请求
- 5 当拿着叉子的哲学家收到了请求，如果筷子是干净的，则他不放下筷子，如果是脏的，则他把筷子擦干净并放下筷子
- 6 哲学家吃完的时候，筷子变脏



Lecture 05: Searching & Model Checking

Crossing the bridge

- The four components of U2 are in a tight situation. Their concert starts in 17 minutes and in order to get to the stage they must first cross an old bridge through which only a maximum of two persons can walk over at the same time.
- It is already dark and, because of the bad condition of the bridge, to avoid falling into the darkness it is necessary to cross it with the help of a ashlight. Unfortunately, they only have one.
- Knowing that Bono, Edge, Adam, and Larry take 1, 2, 5, and 10 minutes, respectively, to cross the bridge, is there a way that they can make it to the concert on time?

Formalization of the game

- The current state of the group can be represented by a multiset (a term of sort Group below) consisting of performers, the ashlight, and a watch to keep record of the time.
- The ashlight and the performers have a Place associated to them, indicating whether their current position is to the left or to the right of the bridge.
- Each performer, in addition, also carries the time it takes him to cross the bridge.
- In order to change the position from left to right and vice versa, we use an auxiliary operation changePos.
- The traversing of the bridge is modeled by two rewrite rules: the first one for the case in which a single person crosses it, and the second one for when there are two.

Maude Code

```
1 mod U2 is
2
3 protecting NAT .
4
5 sorts Performer Object Group Place .
6 subsorts Performer Object < Group .
7
8 ops left right : -> Place .
9
10 op flashlight : Place -> Object .
11 op watch : Nat -> Object .
12 op performer : Nat Place -> Performer .
13 op __ : Group Group -> Group [assoc comm] .
14
15 op changePos : Place -> Place .
16
17 eq changePos(left) = right .
18 eq changePos(right) = left .
```

Maude Code

```
1 op initial : -> Group .
2
3 eq initial
4 = watch(0) flashlight(left) performer(1, left)
5   performer(2, left) performer(5, left) performer(10, left) .
6
7 var P : Place .
8 vars M N N1 N2 : Nat .
9
10 rl [one-crosses] :
11   watch(M) flashlight(P) performer(N, P)
12   =>
13   watch(M + N) flashlight(changePos(P)) performer(N, changePos(P)) .
14
15 crl [two-cross] :
16   watch(M) flashlight(P) performer(N1, P) performer(N2, P)
17   =>
18   watch(M + N1) flashlight(changePos(P))
19   performer(N1, changePos(P))
20   performer(N2, changePos(P))
21 if N1 > N2
```

Searching for the solution

- A solution can be found by looking for a state in which all performers and the ashlight are to the right of the bridge.
- The search command is invoked with a such that clause that allows to introduce a condition that solutions have to fulfill, in our example, that the total time is less than or equal to 17 minutes:

```
1 Maude> search [1] initial =>* flashlight(right) watch(N:Nat)
2     performer(1, right) performer(2, right)
3     performer(5, right) performer(10, right)
4     such that N:Nat <= 17 .
5
6 Solution 1 (state 402)
7 N --> 17
```

Searching for the solution

- The solution takes exactly 17 minutes (a happy ending after all!) and the complete sequence of appropriate actions can be shown with the command

```
1 Maude> show path 402 .
```
- After sorting out the information, it becomes clear that Bono and Edge have to be the first to cross. Then Bono returns with the flashlight, which gives to Adam and Larry. Finally, Edge takes the flashlight back to Bono and they cross the bridge together for the last time.
- Note that, in order for the search command to stop, we need to tell Maude to look only for one solution. Otherwise, it will continue exploring all possible combinations, increasingly taking a larger amount of time, and it will never end.

Model Checking

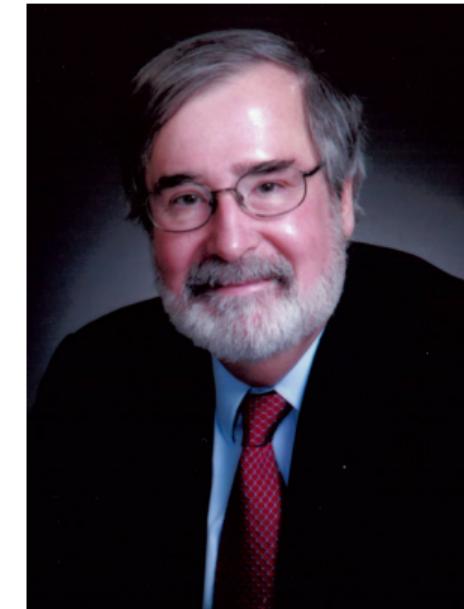
- Two levels of specification:
 - a system specification level, provided by the rewrite theory specified by that system module, and
 - a property specification level, given by some properties that we want to state and prove about our module.
- Temporal logic allows specification of properties such as safety properties (ensuring that something bad never happens) and liveness properties (ensuring that something good eventually happens), related to the infinite behavior of a system.
- Maude includes a model checker to prove properties expressed in linear temporal logic (LTL).

Model checking (模型检测)

关于模型检测和 Edmund Clarke：

模型检测是一种利用数学方法自动验证计算机软件和硬件系统是否满足一定性质的技术。由 Edmund Clarke 教授于 1981 年在其博士论文中首次提出。

- 1 杜克大学数学系硕士
康奈尔大学计算机系博士
- 2 卡耐基梅隆大学计算机系教授
- 3 模型检测创始人
- 4 2007 年图灵奖获得者



What is model checking?

$$\mathcal{M} \models \psi$$

- \mathcal{M} : the model of system
- ψ : a formula specifying the property to be verified

Modeling systems as a Kripke structure \mathcal{K}

Kripke structure

A Kripke structure is a triple : $\langle S, \rightarrow_{\mathcal{K}}, L \rangle$

- S : the set of **states**
- $\rightarrow_{\mathcal{K}}$: a **total** binary relation on S ,
- L : the labeling function: $L : S \rightarrow 2^{AP}$ such that $\forall a \in AP, \forall s \in S. a \in L(s)$ if and only if a holds in s .

Remark: a binary relation b on a set A is total if and only if for each $a \in A$, there exists a a' in A such that $(a, a') \in b$.

Formalizing system properties as logical formula

ψ : logical formula

In computer science or mathematics, there are a number of logics:

- Propositional logic
- First-order logic
- High-order logic
- LTL (Linear temporal logic)
- CTL (Computation tree logic)

Atomic proposition

An atomic proposition is a statement (or parameterized statement).

Example

Statement: $\phi \triangleq \text{Today is Sunday.}$

- $2016.12.12 \models \phi$: false
- $2016.12.11 \models \phi$: true

LTL (Linear temporal logic)

AP: atomic proposition (原子命题)

- True: $\top \in \text{LTL}(AP)$.
- Atomic propositions: If $P \in AP$, then $P \in \text{LTL}(AP)$.
- Next operator: If $P \in \text{LTL}(AP)$, then $\bigcirc P \in \text{LTL}(AP)$.
- Until operator: If $P_1, P_2 \in \text{LTL}(AP)$, then $P_1 \mathbf{U} P_2 \in \text{LTL}(AP)$.
- Boolean connectives: If $P_1, P_2 \in \text{LTL}(AP)$, then $\neg P_1$ and $P_1 \vee P_2$ are in $\text{LTL}(AP)$.

其他算子:

- \Box : Global operator
- \Diamond : Eventual operator

LTL 语义

路径 $\pi = s_0, s_1, s_2, \dots$, where $s_0 \in I$ and $(s_i, s_{i+1}) \in T$ for each $i = (0, 1, \dots)$.

- $(\mathcal{K}, \pi) \models \top$ is true
- $(\mathcal{K}, \pi) \models P \iff (\mathcal{K}, \pi[0]) \models P$ if $P \in AP$.
- $(\mathcal{K}, \pi) \models \bigcirc P \iff (\mathcal{K}, \pi(1)) \models P$.
- $(\mathcal{K}, \pi) \models P_1 \mathbf{U} P_2 \iff \exists n \in N. (\mathcal{K}, \pi(n)) \models P_2 \wedge (\mathcal{K}, \pi(i)) \models P_1$ with $i = (0, 1, \dots, n - 1)$
- $(\mathcal{K}, \pi) \models \Box P_1 \iff \forall n \geq 0. (\mathcal{K}, \pi(n)) \models P_1$
- $(\mathcal{K}, \pi) \models \Diamond P_1 \iff \exists n \geq 0. (\mathcal{K}, \pi(n)) \models P_1$

$\pi(n)$ 表示第 n 个状态及之后的状态组成的子路径

$\pi[n]$ 表示第 n 个状态

Modeling systems in Maude as a rewrite theory

A rewrite theory $\mathcal{R} \triangleq \langle \Sigma, E \cup A, R \rangle$

Recall that T_Σ is the set of all the terms that are constructed by operators in Σ .

Let **State** be a sort to formalize system states.

$T_{\Sigma, \text{State}}$ be the terms of sort **State**.

State transition in rewrite theory

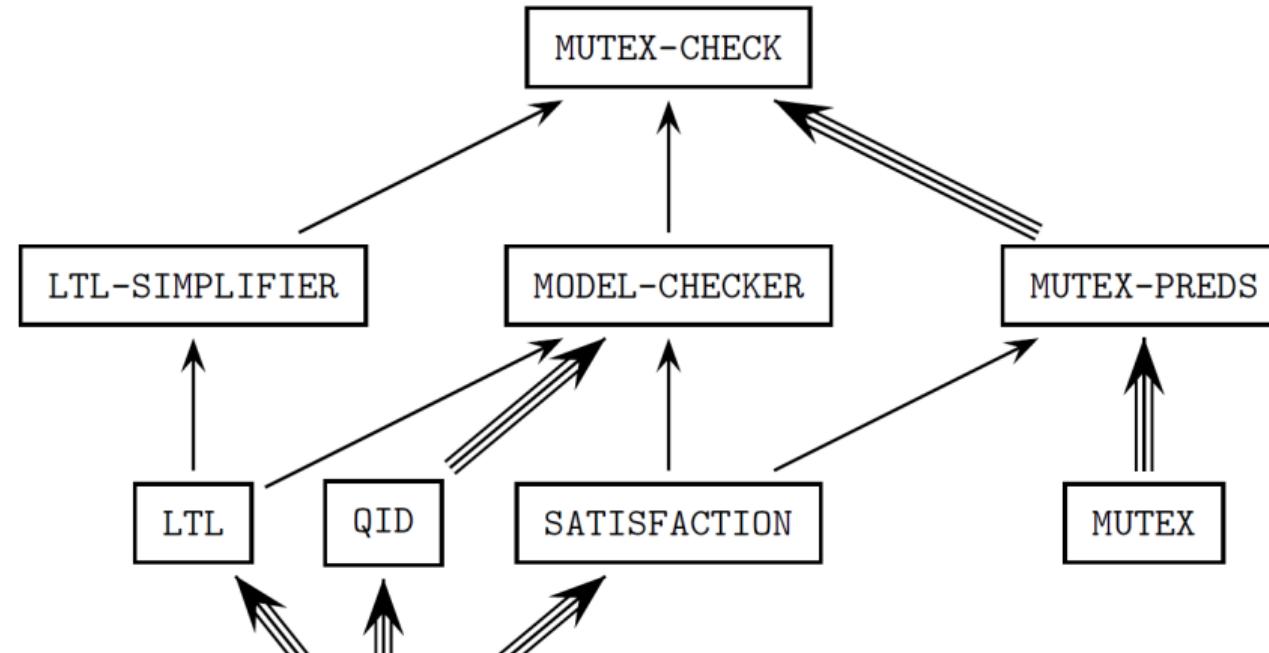
For two arbitrary terms t, t' in $T_{\Sigma, \text{State}}$, there is a transition from the state represented by t to the one represented by t' if there exists a rewrite rule r in R such that $t \rightarrow_r^1 t'$.

Remark: $t \rightarrow_{r:t_1 \rightarrow t_2}^1 t'$: There exist C, σ such that $t = C[\sigma(t_1)]$ and $t' = C[\sigma(t_2)]$.

A rewrite rule represents a SET $\llbracket r \rrbracket$ of state transitions, i.e., $\llbracket r \rrbracket \subseteq \text{State} \times \text{State}$

在 Maude 中描述状态系统

- 利用 sort 定义状态类型 State
- 利用规则定义状态之间的转移关系



互斥算法

Example

- 假设系统中有两个进程 a 和 b,
- 假设系统中有两个 token, \$ 和 *
- 规则
 - 如果系统中有一个 token \$ 且 a 在等待, 那么 a 移除 \$ 且进入临界区
 - 如果系统中有一个 token * 且 b 在等待, 那么 b 移除 * 且进入临界区
 - 如果 a 在临界区, 那么他可以出来并发出一个 token *
 - 如果 b 在临界区, 那么他可以出来并发出一个 token \$

$$\$ \ a \ [] \Rightarrow [a] \tag{1}$$

$$* \ b \ [] \Rightarrow [b] \tag{2}$$

$$[a] \Rightarrow [] \ a \ * \tag{3}$$

$$[b] \Rightarrow [] \ b \ \$ \tag{4}$$

在 Maude 中描述互斥算法

```
1 mod MUTEX is
2   sorts Name Mode Proc Token Conf .
3   subsorts Token Proc < Conf .
4
5   op none : -> Conf [ctor] .
6   op __ : Conf Conf -> Conf [ctor assoc comm id: none] .
7   ops a b : -> Name [ctor] .
8   ops wait critical : -> Mode [ctor] .
9   op [_,_] : Name Mode -> Proc [ctor] .
10  ops * $ : -> Token [ctor] .
11
12  rl [a-enter] : $ [a, wait] => [a, critical] .
13  rl [b-enter] : * [b, wait] => [b, critical] .
14  rl [a-exit] : [a, critical] => [a, wait] * .
15  rl [b-exit] : [b, critical] => [b, wait] $ .
16 endm
```

在 Maude 中定义原子命题

A predefined module: SATISFACTION

```
1 fmod SATISFACTION is
2   protecting BOOL .
3   sorts State Prop .
4   op _|=_ : State Prop -> Bool [frozen (1 2)] .
5 endfm
```

在 Maude 中定义原子命题

```
1 mod MUTEX-PREDS is
2   protecting MUTEX .
3   including SATISFACTION .
4   subsort Conf < State .
5
6   op crit : Name -> Prop .
7   op wait : Name -> Prop .
8
9   var N : Name .
10  var C : Conf .
11  var P : Prop .
12
13 eq [N, critical] C |= crit(N) = true .
14 eq [N, wait] C |= wait(N) = true .
15 eq C |= P = false [owise] .
16 endm
```

Formalizing LTL Formula in Maude

A predefined module: LTL

```
1 fmod LTL is
2   protecting BOOL .
3   sorts Formula .
4   ops True False : -> Formula [ctor format (g o)] .
5   op ~_ : Formula -> Formula [ctor prec 53 format (r o d)] .
6   op _/\_ : Formula Formula -> Formula [comm ctor prec 55 gather (E e) format (
7     d r o d)] .
8   op _\/_ : Formula Formula -> Formula [comm ctor prec 59 gather (E e) format (
9     d r o d)] .
10  op O_ : Formula -> Formula [ctor prec 53 format (r o d)] .
11  op _U_ : Formula Formula -> Formula [ctor prec 63 format (d r o d)] .
12  op _R_ : Formula Formula -> Formula [ctor prec 63 format (d r o d)] .
13  op _->_ : Formula Formula -> Formula [prec 65 gather (e E) format (d r o d)] .
14 .
```

Formalizing LTL Formula in Maude

A predefined module: LTL

```

1  op _<->_ : Formula Formula -> Formula [prec 65 format (d r o d)] .
2  op <>_ : Formula -> Formula [prec 53 format (r o d)] .
3  op `[_` : Formula -> Formula [prec 53 format (r d o d)] .
4  op _W_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
5  op _|->_ : Formula Formula -> Formula [prec 63 format (d r o d)] .
6  op _=>_ : Formula Formula -> Formula [prec 65 gather (e E) format (d r o d)]
7 .
8  op _<=>_ : Formula Formula -> Formula [prec 65 format (d r o d)] .
9  vars f g : Formula .
10 eq f -> g = ~ f \vee g .
11 eq f <-> g = (f -> g) /\ (g -> f) .
12 eq <> f = True U f .
13 eq [] f = False R f .
14 eq f W g = (f U g) \vee [] f .
15 eq f |-> g = [] (f -> (<> g)) .
16 eq f => g = [] (f -> g) .
17 eq f <=> g = [] (f <-> g) .
18 eq ~ True = False .
19 eq ~ False = True .

```

在 Maude 中做模型检测

```
1 mod MUTEX-CHECK is
2   protecting MUTEX-PREDS .
3   including MODEL-CHECKER .
4   including LTL-SIMPLIFIER .
5
6 ops initial1 initial2 : -> Conf .
7
8 eq initial1 = $ [a, wait] [b, wait] .
9 eq initial2 = * [a, wait] [b, wait] .
10 endm
11
12 Maude> red modelCheck(initial1, [] ~(crit(a) /\ crit(b))) .
13 reduce in MUTEX-CHECK :
14 modelCheck(initial1, []~ (crit(a) /\ crit(b))) .
15 result Bool: true
16
17 Maude> red modelCheck(initial2, [] ~(crit(a) /\ crit(b))) .
18 reduce in MUTEX-CHECK :
19 modelCheck(initial2, []~ (crit(a) /\ crit(b))) .
20 result Bool: true
```

在 Maude 中做模型检测

```
1 验证性质：如果 a 在等待，那么他一定能进入到临界区
2 Maude> red modelCheck(initial1, ([]<> wait(a)) -> ([]<> crit(a))) .
3 reduce in MUTEX-CHECK :
4 modelCheck(initial1, []<> wait(a) -> []<> crit(a)) .
5 result Bool: true
6
7 验证性质：如果 b 在等待，那么他一定能进入到临界区
8 Maude> red modelCheck(initial1, ([]<> wait(b)) -> ([]<> crit(b))) .
9 reduce in MUTEX-CHECK :
10 modelCheck(initial1, []<> wait(b) -> []<> crit(b)) .
11 result Bool: true
12
13 Maude> red modelCheck(initial2, ([]<> wait(a)) -> ([]<> crit(a))) .
14 reduce in MUTEX-CHECK :
15 modelCheck(initial2, []<> wait(a) -> []<> crit(a)) .
16 result Bool: true
17
18 Maude> red modelCheck(initial2, ([]<> wait(b)) -> ([]<> crit(b))) .
19 reduce in MUTEX-CHECK :
20 modelCheck(initial2, []<> wait(b) -> []<> crit(b)) .
21 result Bool: true
```

不成立的情况

A counterexample is provided to witness the violation.

```
1 验证 a 始终都在临界区
2 red modelCheck(initial1, [] crit(a)) .
3
4 result ModelCheckResult:
5 counterexample(nil,
6 {$ [a,wait] [b,wait], 'a-enter'}
7 {[a,critical] [b,wait], 'a-exit'}
8 {* [a,wait] [b,wait], 'b-enter'}
9 {[a,wait] [b,critical], 'b-exit})
```

Crossing the river

Formalizing the problem in Maude

- The shepherd and his belongings are represented as objects with an attribute indicating the side of the river in which each is located.
- Constants `left` and `right` represent the two sides of the river.
- Operation `change` is used to modify the corresponding attributes.
- **Rules** represent the ways of crossing the river that are allowed by the capacity of the boat.
- Properties define the good and bad states:
 - `success` characterizes the state in which the shepherd and his belongings are in the other side,
 - `disaster` characterizes the states in which some eating takes place.

Maude system module of the problem

```
1 mod RIVER-CROSSING is
2
3   sorts Side Group .
4
5   ops left right : -> Side [ctor] .
6   op change : Side -> Side .
7   eq change(left) = right .
8   eq change(right) = left .
9
10 ops s w l c : Side -> Group [ctor] .
11 op __ : Group Group -> Group [ctor assoc comm] .
12 var S : Side .
13
14 rl [shepherd] : s(S) => s(change(S)) .
15 rl [wolf] : s(S) w(S) => s(change(S)) w(change(S)) .
16 rl [lamb] : s(S) l(S) => s(change(S)) l(change(S)) .
17 rl [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .
18 endm
```

Formalizing state predicates

```
1 mod RIVER-CROSSING-PROP is
2   protecting RIVER-CROSSING .
3
4   including MODEL-CHECKER .
5
6   subsort Group < State .
7   op initial : -> Group .
8   eq initial = s(left) w(left) l(left) c(left) .
9   ops disaster success : -> Prop .
10
11  vars S S' S'' : Side .
12
13  ceq (w(S) l(S) s(S') c(S'') |= disaster) = true if S =/= S' .
14  ceq (w(S'') l(S) s(S') c(S) |= disaster) = true if S =/= S' .
15  eq (s(right) w(right) l(right) c(right) |= success) = true .
16 endm
```

Formalizing system properties using LTL

- The model checker only returns paths that are counterexamples of properties.
- To find a safe path we need to find a formula that somehow expresses the negation of the property we are interested in: a counterexample will then witness a safe path for the shepherd.
- If no safe path exists, then it is true that whenever success is reached a disastrous state has been traversed before:

```
| <> success -> (<> disaster /n ((~ success) U disaster))
```

Note that this formula is equivalent to the simpler one

```
| <> success -> ((~ success) U disaster)
```

- A counterexample to this formula is a safe path, completed so as to have a cycle.

Model Checking Result

```
1 Maude> red modelCheck(initial,  
2 <> success -> (<> disaster /n ((~ success) U disaster))) .  
3  
4 result ModelCheckResult: counterexample(  
5 {s(left) w(left) l(left) c(left), 'lamb}  
6 {s(right) w(left) l(right) c(left), 'shepherd}  
7 {s(left) w(left) l(right) c(left), 'wolf}  
8 {s(right) w(right) l(right) c(left), 'lamb}  
9 {s(left) w(right) l(left) c(left), 'cabbage}  
10 {s(right) w(right) l(left) c(right), 'shepherd}  
11 {s(left) w(right) l(left) c(right), 'lamb}  
12 {s(right) w(right) l(right) c(right), 'lamb}  
13 {s(left) w(right) l(left) c(right), 'shepherd}  
14 {s(right) w(right) l(left) c(right), 'wolf}  
15 {s(left) w(left) l(left) c(right), 'lamb}  
16 {s(right) w(left) l(right) c(right), 'cabbage}  
17 {s(left) w(left) l(right) c(left), 'wolf},  
18 {s(right) w(right) l(right) c(left), 'lamb}  
19 {s(left) w(right) l(left) c(left), 'lamb})
```

总结 (tip of the iceberg)

模型检测

- 完全自动化
- 如果性质不满足，可以得到反例
- 可以验证 safety property 和 liveness property
- 模型检测工具：SPIN, SMV, NuSMV, PAT 等

关于更多模型检测的知识，请参考：

《Model checking》 Edmund Clarke, MIT Press, 1999

Lecture 06: Reflection and Meta-Programming in Maude

Reflection

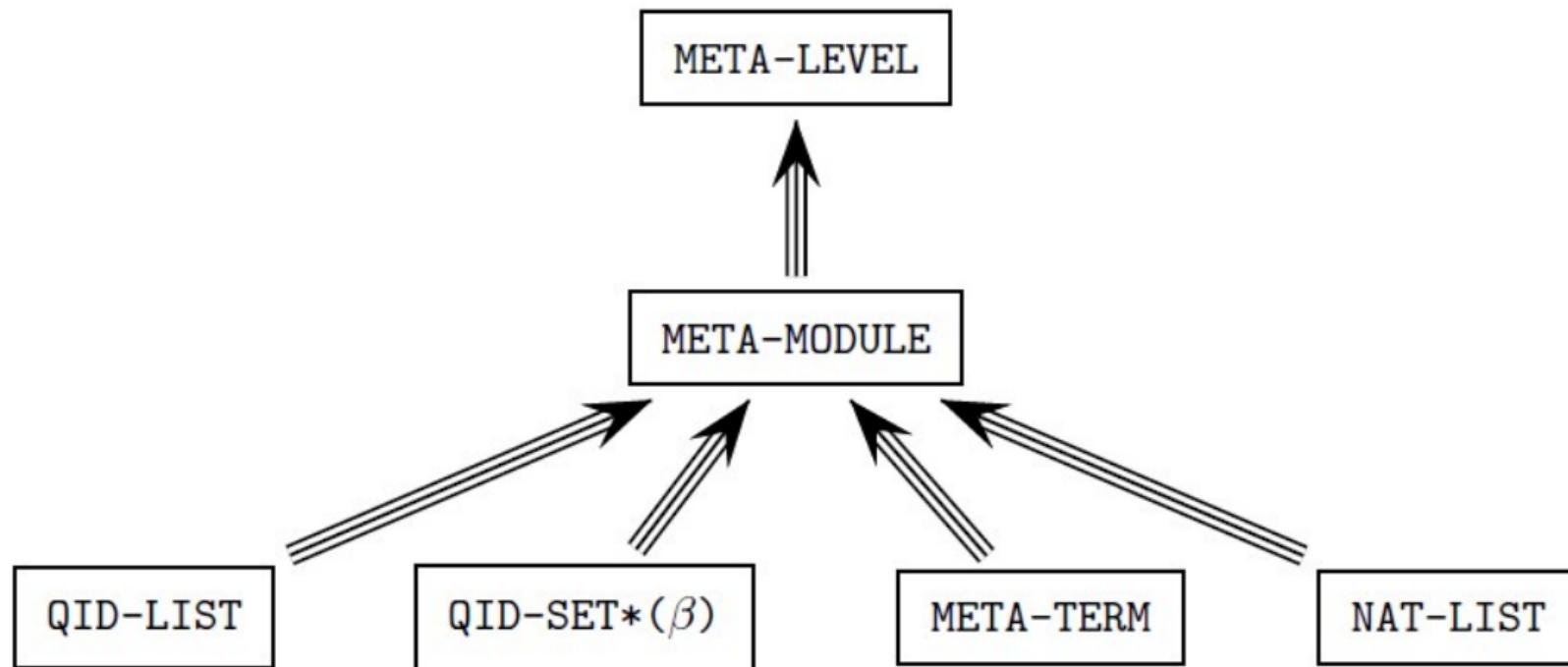
- Rewriting logic is **reflective** in that there is a finitely presented rewrite theory \mathcal{U} that is **universal** in the sense that:
 - we can represent any finitely presented rewrite theory \mathcal{R} and any terms t, t' in \mathcal{R} as **terms** $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in \mathcal{U} ,
 - The following equivalence holds:

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle$$

- Since \mathcal{U} is representable in itself, we get a reflective tower

$$\begin{array}{c} \mathcal{R} \vdash t \longrightarrow t' \\ \Updownarrow \\ \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle \\ \Updownarrow \\ \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \longrightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \end{array}$$

Maude's meta-level



Maude's meta-level

In Maude, key functionality of the universal theory \mathcal{U} has been efficiently implemented in the functional module META-LEVEL:

- Maude **terms** are reified as elements of a data type **Term** in the module META-TERM;
- Maude modules are reified as terms in a data type **Module** in the module META-MODULE;
- operations **upModule**, **upTerm**, **downTerm**, and others allow moving between reflection levels;
- the process of reducing a term to canonical form using Maude's reduce command is meta-represented by a built-in function **metaReduce**;
- the processes of rewriting a term in a system module using Maude's rewrite and frewrite commands are meta-represented by built-in functions **metaRewrite** and **metaFrewrite**;

Maude's meta-level

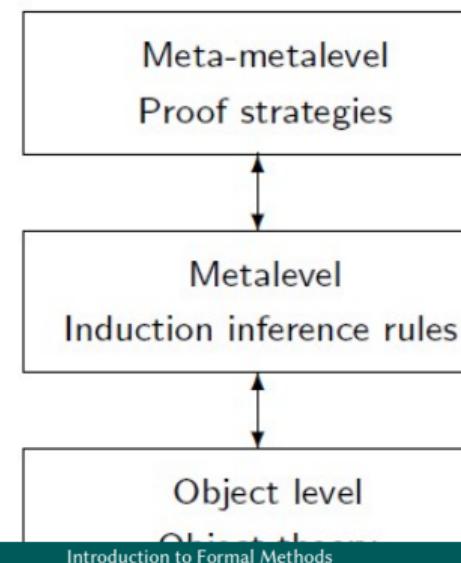
- the process of **applying a rule** of a system module at the **top** of a term is meta-represented by a built-in function `metaApply`;
- the process of applying a rule of a system module at **any** position of a term is meta-represented by a built-in function `metaXapply`;
- the process of **matching two terms** is reified by built-in functions `metaMatch` and `metaXmatch`;
- the process of **searching for a term** satisfying some conditions starting in an initial term is reified by built-in functions `metaSearch` and `metaSearchPath`; and
- parsing and **pretty-printing** of a term in a module, as well as key sort operations such as comparing sorts in the subsort ordering of a signature, are also meta-represented by corresponding built-in functions.

Meta-programming

- **Programming at the metalevel:** the metalevel equations and rewrite rules operate on representations of lower-level rewrite theories.
- Reflection makes possible many advanced meta-programming applications, including
 - user-definable **strategy languages**,
 - language extensions by new module composition operations,
 - development of **theorem proving tools**, and
 - definition of **translations between languages** or logics within rewriting logic.
- Theorem provers and other formal tools have **underlying inference systems that can be naturally specified and prototyped in rewriting logic**. Furthermore, the strategy aspects of such tools and inference systems can then be specified by rewriting strategies.

Developing theorem proving tools

- Theorem-proving tools have a very simple reflective design in Maude.
- The inference system itself may perform theory transformations, so that the theories themselves must be treated as data.
- We need strategies to guide the application of the inference rules.
- Example: Inductive Theorem Prover (ITP).



Full Maude

- The systematic and efficient use of reflection through its predefined META-LEVEL module makes Maude remarkably **extensible** and powerful.
- Full Maude is **an extension of Maude, written in Maude itself**, that endows the language with an even more powerful and extensible module algebra of parameterized modules and module composition operations, including parameterized views.
- Full Maude also provides special syntax for object-oriented modules supporting object-oriented concepts such as objects, messages, classes, and multiple class inheritance.

Full Maude

- Full Maude itself can be used as a basis for further extensions, by adding new functionality.
- Full Maude becomes a common infrastructure on top of which one can build other tools:
 - Church-Rosser and coherence checkers for Maude
 - declarative debuggers for Maude, for wrong and missing answers
 - Real-Time Maude tool for specifying and analyzing real-time systems
 - MSOS tool for modular structural operational semantics
 - Maude-NPA for analyzing cryptographic protocols
 - strategy language prototype

Lecture 07: A Mini Language in Maude

Minila: Mini-Language

语法 (Syntax)

$$\begin{aligned}
 s ::= & \quad v(i) := e & i \in \mathbb{N} \\
 | & \text{ if } e \text{ then } s \text{ else } s \text{ fi} \\
 | & \text{ while } e \text{ do } s \text{ od} \\
 | & s \text{ ; ; } s \\
 e ::= & \quad v(i) & i \in \mathbb{N} \\
 | & v & v \in \mathbb{N} \cup \{\text{true}, \text{false}\} \\
 | & e \star e \\
 \star ::= & \quad ++ \mid -- \mid ** \mid // \mid !> \mid <!
 \end{aligned}$$

语义 (semantics): 与 C 和 Java 语言的语义相同。

例子

```
1 v(0) := 0 ;;
2 v(1) := 10 ;;
3 while v(1) !> 0 do
4   v(0) := v(0) ++ v(1) ;;
5   v(1) := v(1) -- 1
6 od
```

问题：如何执行这段程序？

执行程序的条件

- 1 机器：用于执行机器指令
- 2 机器的指令：被机器识别并执行
- 3 编译器：把程序翻译成机器指令

机器和机器语言

机器：

- 1 寄存器：存放各变量的值
- 2 栈：存放指令

机器语言：

push | add | mul | n

例子

计算下面的表达式：

10 ** (20 + 30)

翻译成机器指令：

指令 : $\overset{pc}{\underset{0}{\text{push}}}; \underset{1}{10}; \underset{2}{\text{push}}; \underset{3}{20}; \underset{4}{\text{push}}; \underset{5}{30}; \underset{6}{\text{add}}; \underset{7}{\text{mul}}; \underset{\text{nil}}{\text{nil}}$

执行

执行上面的指令：

code	pc	stack
0 push; 10; 1 push; 20; 2 push; 30; 3 add; 4 mul; 5 nil 6	0	empty
0 push; 10; 1 push; 20; 2 push; 30; 3 add; 4 mul; 5 nil 6	2	10 # empty
0 push; 10; 1 push; 20; 2 push; 30; 3 add; 4 mul; 5 nil 6	4	20 # 10 # empty
0 push; 10; 1 push; 20; 2 push; 30; 3 add; 4 mul; 5 nil 6	6	30 # 20 # 10 # empty
0 push; 10; 1 push; 20; 2 push; 30; 3 add; 4 mul; 5 nil 6	7	50 # 10 # empty
0 push; 10; 1 push; 20; 2 push; 30; 3 add; 4 mul; 5 nil 6	8	500 # empty
500		

形式化定义虚拟机的执行

虚拟机可以看做一个函数: vm

参数:

- 1 一串机器指令;
- 2 当前执行的位置;
- 3 一个栈

返回: 一个数值

形式化定义虚拟机的执行

定义指令：

```
1 fmod INSTRUCTION is
2   including NAT .
3   sort Instruction .
4   subsort Nat < Instruction .
5   ops push add mul : -> Instruction .
6 endfm
```

形式化定义虚拟机的执行

定义指令串：

```
1 fmod INS-LIST is
2   including INSTRUCTION .
3   sort InsList .
4   subsort Instruction < InsList .
5   op nil : -> InsList .
6   op _;_ : InsList InsList -> InsList [assoc id: nil] .
7   op getIns : InsList Nat -> Instruction .
8   var IL : InsList .
9   var I : Instruction .
10  var N : Nat .
11  eq getIns(I ; IL, 0) = I .
12  ceq getIns(I ; IL, N) = getIns(IL, sd(N,1)) if N > 0 .
13  op len : InsList -> Nat .
14  eq len(nil) = 0 .
15  eq len(I ; IL) = len(IL) + 1 .
16 endfm
```

形式化定义虚拟机的执行

定义栈：

```
1 fmod STACK is
2   including NAT .
3   sort Stack .
4   subsort Nat < Stack .
5   op empty : -> Stack .
6   op _#_ : Stack Stack -> Stack [assoc id: empty] .
7 endfm
```

形式化定义虚拟机的执行

定义虚拟机：

```

1 fmod VIRTUAL-MACHINE is
2   including INS-LIST .
3   including STACK .
4   op vm : InsList Nat Stack -> Nat .
5   var IL : InsList .
6   var S : Stack .
7   vars N N1 N2 : Nat .

8
9   ceq vm(IL, N , S) = vm(IL, N + 2 , getIns(IL, N + 1) # S)
10  if getIns(IL,N) == push .
11  ceq vm(IL, N , N1 # N2 # S) = vm(IL, N + 1 , N1 * N2 # S)
12  if getIns(IL,N) == mul .
13  ceq vm(IL, N , N1 # N2 # S) = vm(IL, N + 1 , N1 + N2 # S)
14  if getIns(IL,N) == add .
15  ceq vm(IL, N , N1 # S) = N1
16  if len(IL) == N .
17 endfm

```

形式化定义虚拟机的执行

运行一次执行：

```
1 Maude> red vm(push ; 10 ; push ; 20 ; push ; 30 ; add ; mul ; nil,0,empty) .  
2 result NzNat: 500
```

从表达式到机器指令串的翻译（编译）

从表达式到机器指令串的翻译（编译）

10 ** (20 ++ 30)

编译规则：

$\text{compile}(e_1 ** e_2) \Rightarrow \text{compile}(e_1); \text{compile}(e_2); \text{mul}$
 $\text{compile}(e_1 ++ e_2) \Rightarrow \text{compile}(e_1); \text{compile}(e_2); \text{add}$
 $\text{compile}(n) \Rightarrow \text{push}; n$

编译的实现

表达式的定义：

```
1 fmod SYNTAX is
2   extending NAT .
3   sort ExNat .
4   subsort Nat < ExNat .
5   ops t f : -> ExNat .
6
7   sort Exp .
8   subsort ExNat < Exp .
9
10  op v(_) : Nat -> Exp .
11  op _*_ : Exp Exp -> Exp .
12  op _++_ : Exp Exp -> Exp .
13  op _--_ : Exp Exp -> Exp .
14  op _//_ : Exp Exp -> Exp .
15  op _!>_ : Exp Exp -> Exp .
16  op _<!_ : Exp Exp -> Exp .
17 endfm
```

编译的实现

编译函数的定义：

```

1 fmod COMPILE-EXP is
2   including INS-LIST .
3   including SYNTAX .
4
5   op compile : Exp -> InsList .
6   vars E1 E2 : Exp .
7   var N : Nat .
8   eq compile(E1 ** E2) = compile(E1) ; compile(E2) ; mul .
9   eq compile(E1 ++ E2) = compile(E1) ; compile(E2) ; add .
10  eq compile(N) = push ; N .
11 endfm

```

运行实例：

```

1 Maude> red compile(10 ** (20 ++ 30)) .
2 result IntList: push ; 10 ; push ; 20 ; push ; 30 ; add ; mul

```

编译的实现

带编译功能的虚拟机：

```
1 fmod VM-COM is
  including VIRTUAL-MACHINE .
  including COMPILE-EXP .
4 endfm
```

运行实例：

```
1 Maude> red vm(compile(10 ** (20 ++ 30)), 0, empty) .
2 result NzNat: 500
```

思考

下面的程序对应的虚拟机指令是什么，如何实现？

```
1 v(0) := 0 ;;
2 v(1) := 10 ;;
3 while v(1) !> 0 do
4   v(0) := v(0) ++ v(1) ;;
5   v(1) := v(1) -- 1
6 od
```

存储变量值的设备：堆 (heap)

如何编译: $v(0) := 0$?

怎么样表示堆?

堆作为映射表

$(x_1 \mapsto v_1), \dots, (x_n \mapsto v_n)$

堆上的操作:

- 更新 (update),
 - 参数: 堆, 变量, 值
 - 返回值: 更新后的堆
- 获取 (value)
 - 参数: 堆, 变量
 - 返回值: 变量在堆中的值

堆 (heap) 的实现

```

1 fmod HEAP is
2   including SYNTAX .
3   sort Heap .
4
5   op `(_|->_)` : Var Nat -> Heap .
6   op empty : -> Heap .
7   op __ : Heap Heap -> Heap [assoc comm id: empty] .
8
9   op update : Heap Var Nat -> Heap .
10  op value : Heap Var -> Nat .
11  vars N1 N2 : Nat .
12  vars V1 V2 : Nat .
13  var M1 : Heap .
14  eq update(empty, v(N1), V1) = (v(N1) |-> V1) .
15  eq update((v(N1) |-> V1) M1, v(N2), V2) =
16    (if N1 == N2 then (v(N1) |-> V2) M1 else
17     (v(N1) |-> V1) update(M1, v(N2), V2) fi) .
18  eq value(empty, v(N1)) = 0 .
19  eq value((v(N1) |-> V1) M1, v(N2)) =
20    (if N1 == N2 then V1 else value(M1, v(N2)) fi) .
21 endfm

```

机器和机器语言

机器：

- 1 寄存器：存放各变量的值
- 2 栈：存放数据
- 3 堆：存放变量

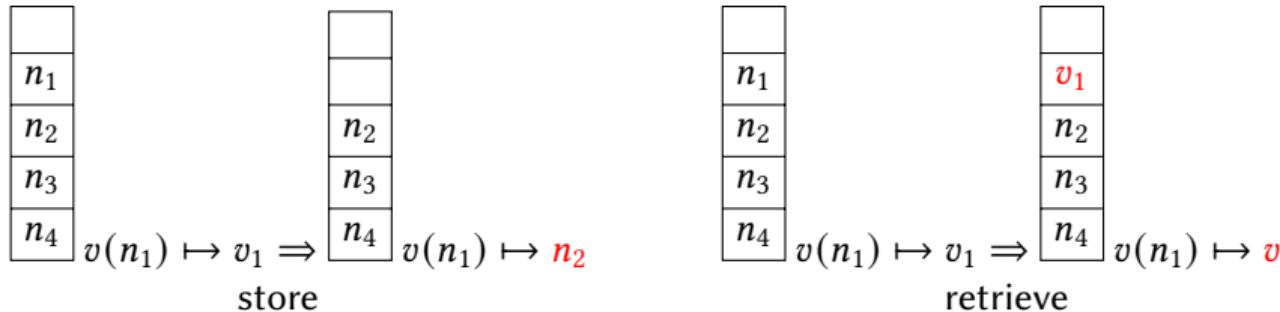
机器语言：

堆操作机器指令

push | add | mul | n | $\overbrace{store \mid retrieve}$

机器指令 store, retrieve 对应的操作

- store: 栈顶的值存放在栈顶第二个值作为堆地址的位置。
- retrieve: 从堆中除去以栈顶的值为地址的值



```

1 --- store
2 ceq vm(IL,N, N1 # N2 # S, M) =
3   vm(IL, N + 1, N2 # S, update(M,v(N1),N2))
4   if getIns(IL,N) = store .
5
6 --- retrieve
7 ceq vm(IL,N, N1 # S, M) =
8   vm(IL, N + 1, value(M,v(N1)) # S, M)
9   if getIns(IL,N) = retrieve .

```

从表达式到机器指令串的翻译（编译）

编译：

```

1      v(0)  := 10 ; ;
2      v(0)  := v(0) ** v(0)

```

编译规则：

$$\text{compile}(v(n) := e_1) = \text{compile}(e_1); \text{push}; n; \text{store}.$$

$$\text{compile}(v(n)) = \text{push}; n; \text{retrieve}.$$

$$\text{compile}(s_1;; s_2) = \text{compile}(s_1); \text{compile}(s_2)$$

编译的实现：

```

1 eq compile(v(N) := E1) = compile(E1) ; push ; N ; store .
2 eq compile(v(N)) = push ; N ; retrieve .
3 eq compile(S1 ;; S2) = compile(S1) ; compile(S2) .

```

运行实例

Example (Example 1)

```
1 v(0) := 10 ** (20 ++ 30)
```

```
1 red compile(v(0) := 10 ** (20 ++ 30)) .
2 red vm(compile(v(0) := 10 ** (20 ++ 30)), 0, empty, empty) .
```

Example (Example 2)

```
1 v(0) := 10 ** (20 ++ 30);;
2 v(0) := v(0) ++ 1
```

```
1 red compile(v(0) := 10 ** (20 ++ 30) ;; v(0) := v(0) ++ 1) .
2 red vm(compile(v(0) := 10 ** (20 ++ 30) ;; v(0) := v(0) ++ 1), 0, empty, empty) .
```

if 语句的翻译

一个 if 语句的例子

```

1 v(0) := 0 ;
2 v(1) := 10 ;
3 if v(1) !> v(0) then
4   v(1) = v(1) -- v(0)
5 else
6   v(1) = v(0) -- v(1)
7 fi

```

分析：

$v(1) !> v(0) \Rightarrow \text{compile}(v(1) !> v(0))$

$v(1) = v(1) -- v(0) \Rightarrow \text{compile}(v(1) = v(1) -- v(0))$

$v(1) = v(0) -- v(1) \Rightarrow \text{compile}(v(1) = v(0) -- v(1))$

$\text{compile}(v(1) !> v(0)); ??; \text{compile}(v(1) = v(1) -- v(0)); ??; \text{compile}(v(1) = v(0) -- v(1))$

if 语句的翻译

一个 if 语句的例子

```

1 v(0) := 0 ; ;
2 v(1) := 10 ; ;
3 if v(1) !> v(0) then
4 v(1) = v(1) -- v(0)
5 else
6 v(1) = v(0) -- v(1)
7 fi

```

分析：

v(1) !> v(0) => compile(v(1) !> v(0))

v(1) = v(1) -- v(0) => compile(v(1) = v(1) -- v(0))

v(1) = v(0) -- v(1) => compile(v(1) = v(0) -- v(1))

```
compile(v(1) !> v(0));zjump; l1;  $\overbrace{\text{compile(v(1) = v(1) -- v(0))}; \text{jump}; l_2}$   
                  l2
```

if 语句的编译

if 语句编译在 Maude 下的定义：

```

1 eq compile(if E1 then S1 else S2 fi) =
2 compile(E1);
3 zjump ; len(compile(S1)) + 4 ;
4 complie(S1) ;
5 jump ; len(compile(S2)) + 2 ;
6 compile(S2) .

```

或者

```

1 ceq compile(if E1 then S1 else S2 fi) =
2 compile(E1);
3 zjump ; len(L1) + 4 ; L1 ; jump ; len(L2) + 2 ; L2
4 if L1 := complie(S1) /\
5 L2 := complie(S2) .

```

思考：为什么要加 4, 加 2 ?

两条新的机器指令

新增机器指令: **zjump**, **jump**

- 零跳转: **zjump**
- 无条件跳转: **jump**

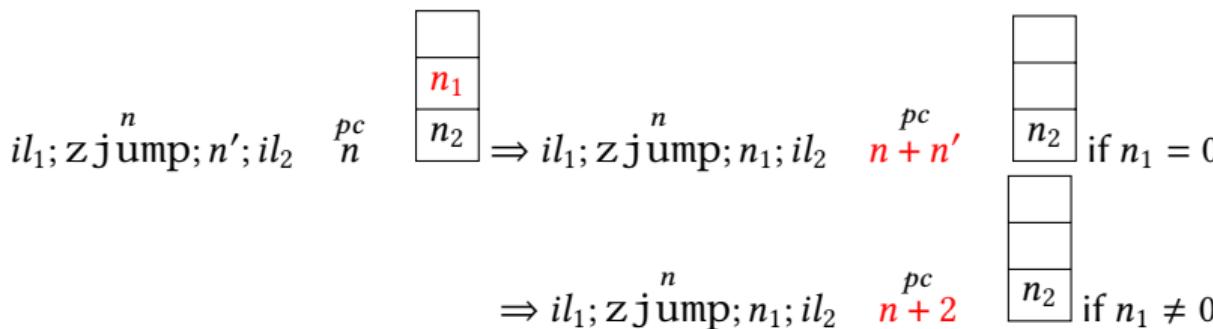
```
1 fmod INSTRUCTION is
2   including NAT .
3   sort Instruction .
4   subsort Nat < Instruction .
5   ops push add mul store retrieve
6   equal unequal greater less zjump jump : -> Instruction .
7 endfm
```

zjump jump 的实现 (虚拟)

```

1 ceq vm(IL, N , N1 # S,M) =
2 vm(IL,
3 (if N1 == 0 then N + getIns(IL, N + 1) else N + 2 fi),
4 S, M)
5 if getIns(IL,N) == zjump .
6
7 ceq vm(IL, N , S,M) =
8 vm(IL, N + getIns(IL, N + 1), S,M)
9 if getIns(IL,N) == jump .

```



Note: $n' = \text{getIns}(N + 1)$

Min Zhang

运行实例

Example (Example 1)

```

1 v(0) := 10 ;;
2 if v(0) !> 0 then
3   v(0) := v(0) ++ 1
4 else
5   v(0) := 0
6 fi

```

```

1 red compile(v(0) := 10 ;; if v(0) !> 0 then v(0) := v(0) ++ 1 else v(0) := 0 fi) .
2 red vm(compile(v(0) := 10 ;; if v(0) !> 0 then v(0) := v(0) ++ 1 else v(0) := 0 fi)
      ,0,empty,empty) .

```

```

1 Result:
2 push ; 10 ; push ; 0 ; store ; push ; 0 ; push ; 0 ; retrieve ;
3 greater ; zjump ; 13 ; push ; 0 ; retrieve ; push ; 1 ; add ;
4 push ; 0 ; store ; jump ; 7 ; push ; 0 ; push ; 0 ; store
5
6 Result: 11

```

while 语句的编译

```
1 while E do  
2 S  
3 do
```

语义：

```
1 if E then  
2 S  
3 while E do  
4 S  
5 do  
6 else  
7 $null$  
8 fi
```

```
1 if E then  
2 S;;  
3 goto 1  
4 else  
5 $null$  
6 fi
```

while 语句的编译

```

1 --- 将 while 循环转换成 if 条件语句，但是会引起不终止
2 --- eq while E1 do S1 od =
3 if E1 then S1 (while E1 do S1 od) else nil fi .
4
5 eq compile(while E1 do S1 od) =
6 compile(E1); zjump ; len(compile(S1)) + 4 ;
7 compile(S1) ; _____ .

```

参考 if 语句的编译：

```

1 eq compile(if E1 then S1 else S2 fi) =
2 compile(E1); zjump ; len(compile(S1)) + 4 ;
3 compile(S1) ; jump ; len(compile(S2)) + 2 ; compile(S2) .

```

while 语句的编译

```

1 --- 将 while 循环转换成 if 条件语句，但是会引起不终止
2 --- eq while E1 do S1 od =
3 if E1 then S1 (while E1 do S1 od) else nil fi .
4
5 eq compile(while E1 do S1 od) =
6 compile(E1); zjump ; len(compile(S1)) + 4 ;
7 compile(S1) ; goto 1 .

```

参考 if 语句的编译：

```

1 eq compile(if E1 then S1 else S2 fi) =
2 compile(E1); zjump ; len(compile(S1)) + 4 ;
3 compile(S1) ; jump ; len(compile(S2)) + 2 ; compile(S2) .

```

while 语句的编译

```

1 --- 将 while 循环转换成 if 条件语句，但是会引起不终止
2 --- eq while E1 do S1 od =
3 if E1 then S1 (while E1 do S1 od) else nil fi .
4
5 eq compile(while E1 do S1 od) =
6 compile(E1); zjump ; len(compile(S1)) + 4 ;
7 compile(S1) ; bjump ; len(compile(S1)) + 2 + len(compile(E1)) .

```

bjump: 回跳机器指令

参考 if 语句的编译：

```

1 eq compile(if E1 then S1 else S2 fi) =
2 compile(E1); zjump ; len(compile(S1)) + 4 ;
3 compile(S1) ; jump ; len(compile(S2)) + 2 ; compile(S2) .

```

bjump 的实现（虚拟）

```
1 ceq vm(IL, N , S,M) =  
2   vm(IL, sd(N, getIns(IL, N + 1)), S,M)  
3 if getIns(IL,N) == bjump .
```

- N: 当前 pc
- sd: 减法运算
- getIns(IL, N + 1): 下一个机器指令, 即要跳转的指令数

while 语句的运行例子

Let P be:

```

1 v(0) := 10 ;;
2 v(1) := 0 ;;
3 while v(0) !> 0 do
4   v(1) := v(1) ++ v(0) ;;
5   v(0) := v(0) -- 1 od;;
6 v(1) := v(1)

```

命令:

```

1 red compile(P) .
2 red vm(compile(P), 0, empty, empty) .

```

运行结果:

```

1 push ; 10 ; push ; 0 ; store ; push ; 0 ; push ; 1 ; store ;push ; 0 ;
2 push ; 0 ; retrieve ; greater ; zjump ; 23 ; push ; 1 ; retrieve;
3 push ; 0 ; retrieve ; add ;push ; 1 ; store ; push ; 0 ; retrieve ;
4 push; 1 ; minus ; push ; 0 ; store ; bjump ; 27 ; push ; 1 ;
5 retrieve ; push ;1 ; store
55

```

Lecture 08: Mathematical Foundation of Maude: Term Rewriting

Quiz

Are the following two equations **terminating**?

$$f(f(x)) = f(g(f(x))) \quad (5)$$

$$g(x) = x \quad (6)$$

Quiz

How about this?

$$0 + x = x \tag{7}$$

$$s(x) + y = s(x + y) \tag{8}$$

Why?

Term

Definition (Term)

Given a signature Σ and a set of variables V , let $T_\Sigma(V)$ be least set of **terms** that satisfy the following two conditions:

- 1 $\forall x \in V, x \in T_\Sigma(V)$
- 2 $\forall t_1, \dots, t_n \in T_\Sigma(V) \text{ and } f^{(n)} \in \Sigma, f(t_1, \dots, t_n) \in T_\Sigma(V).$

Example

In the module **NAT**, some examples of terms are:

- 0, s 0
- X, s s 0 + s Y

Ground term

Ground terms: terms that do not contain variables.

Definition (Ground term)

Ground term are those terms in $T_\Sigma(\emptyset)$, which is also denoted as T_Σ

Definition (Equivalence class of ground terms)

Given an equational theory $\mathcal{E} = (\Sigma, E \cup A)$, for a ground term t let $[t] = \{t' | t =_{E \cup A} t'\}$.

Example

$[s\ s\ 0] = \{s\ s\ 0, 0 + s\ s\ 0, s\ 0 + s\ 0, \dots\}$

Let $T_{\Sigma/E \cup A}$ be the set of equivalent classes of ground terms in T_Σ .

$T_{\Sigma/E \cup A}$ is called the **initial algebra** of \mathcal{E} .

Notations on terms

Definition

Given a term t in $T_{\Sigma}(X)$

- $\mathcal{V}ar(t) = \begin{cases} \{t\} & \text{if } t \in X \\ \mathcal{V}ar(t_1) \cup \dots \cup \mathcal{V}ar(t_n) & \text{if } t = f(t_1, \dots, t_n) \end{cases}$
- $\mathcal{F}un(t) = \begin{cases} \emptyset & \text{if } t \in X \\ \{f\} \cup \mathcal{F}un(t_1) \cup \dots \cup \mathcal{F}un(t_n) & \text{if } t = f(t_1, \dots, t_n) \end{cases}$
- $root(t) = \begin{cases} t & \text{if } t \in X \\ f & \text{if } t = f(t_1, \dots, t_n) \end{cases}$

Example

- $\mathcal{V}ar(s(x) + y) = \{x, y\}$
- $\mathcal{F}un(s(x) + y) = \{s, +\}$
- $root(s(x) + y) = +$

Context and substitution

Definition

- A **context** C is a term in $\mathcal{T}_{\Sigma \cup \{\square\}}(X)$ with exactly one hole \square
- $C[t]$: The term by substituting t for \square in C .
- **substitution**:
 $\sigma : X \rightarrow T_{\Sigma}(X)$ is a substitution if $\text{Dom}(\sigma) = \{x | x \in \mathcal{V}, \sigma(x) \neq x\}$ is finite.
- $t\sigma = \begin{cases} \sigma(t) & \text{if } t \in X \\ f(t_1\sigma, \dots, t_n\sigma) & \text{if } t = f(t_1, \dots, t_n) \end{cases}$

Example

- $C[s(x)] = s(\textcolor{red}{s(x)} + y)$ for $C = s(\square + y)$
- $(s(x) + y)\sigma = s(s(y)) + s(0)$ for $\sigma = \{x \mapsto s(y), y \mapsto s(0)\}$

Match

Definition (Match)

Given two terms t, t' , t' matches t iff there exists a context C , a term t'' and a substitution σ such that $t = C[t'']$ and $t'\sigma = t''$.

Example

`fac(s N)` matches `fac(s s 0)`, where $C = \square$, $t'' = \text{fac}(s\ s\ 0)$, $\sigma = \{\text{N} \mapsto \text{s } 0\}$

It is clear that $t = C[t'']$ and $t'\sigma = t''$.

Matching algorithm

Definition (Problem)

Given two terms s, t , does s match t with context $C = \square$?

Matching algorithm

- 1 $\{s \mapsto t\}$
- 2 Repeat the following rules:

$$\{f(s_1, \dots, s_n) \xrightarrow{} f(t_1, \dots, t_n)\} \cup S \Rightarrow \{s_1 \xrightarrow{} t_1, \dots, s_n \xrightarrow{} t_n\} \cup S \quad (9)$$

$$\{f(s_1, \dots, s_n) \xrightarrow{} g(t_1, \dots, t_n)\} \cup S \Rightarrow \perp \text{ if } f \neq g \quad (10)$$

$$\{f(s_1, \dots, s_n) \xrightarrow{} x\} \cup S \Rightarrow \perp \quad (11)$$

$$\{x \xrightarrow{} t\} \cup S \Rightarrow \perp \text{ if } x \xrightarrow{} t' \in S \text{ with } t \neq t' \quad (12)$$

Example about matching

Example (Example 1)

$x + s(y + z)$ matches $s(y) + s((x + s(0)) + z)$?

$$\begin{aligned} & \{x + s(y + z) \xrightarrow{} s(y) + s((x + s(0)) + z)\} \\ & \Rightarrow \{x \xrightarrow{} s(y), s(y + z) \xrightarrow{} s((x + s(0)) + z)\} \\ & \Rightarrow \{x \xrightarrow{} s(y), y + z \xrightarrow{} (x + s(0)) + z\} \\ & \Rightarrow \{x \xrightarrow{} s(y), y \xrightarrow{} x + s(0), z \xrightarrow{} z\} \end{aligned}$$

Example (Example 2)

$x^- \cdot (x \cdot y)$ matches $(e \cdot x)^- \cdot ((e \cdot e) \cdot x)$?

$$\begin{aligned} & \{x^- \cdot (x \cdot y) \xrightarrow{} (e \cdot x)^- \cdot ((e \cdot e) \cdot x)\} \\ & \Rightarrow \{x^- \xrightarrow{} (e \cdot x)^-, x \cdot y \xrightarrow{} (e \cdot e) \cdot x\} \\ & \Rightarrow \{x \xrightarrow{} c \cdot x, x \cdot y \xrightarrow{} (e \cdot e) \cdot x\} \\ & \Rightarrow \{x \xrightarrow{} c \cdot x, x \xrightarrow{} e \cdot e, y \xrightarrow{} x\} \\ & \Rightarrow \perp \end{aligned}$$

Rewriting

Definition (Rewriting)

Given a term t , an equation $l = r$, if there exists a context C , a term t'' and a substitution σ such that $t = C[t'']$ and $l\sigma = t''$, t is rewritten into $C[r\sigma]$, denoted by $t \rightarrow C[r\sigma]$.

Example

`fac(s N)` matches `fac(s s 0)`, where $C = \square$, $t'' = \text{fac}(s s 0)$, $\sigma = \{\text{N} \mapsto \text{s 0}\}$

By equation $\text{fac}(s N) = (s N) * \text{fac}(N)$, $\text{fac}(s s 0) \rightarrow (s N * \text{fac}(N))\sigma$, which equals to `s s 0 * fac(s 0)`.

Normal form

Definition (Normal form)

Given a term t and a set E of equations, t' is called a **normal form** of t if $t \rightarrow^* t'$ and t' can not be rewritten further by E .

Example

The normal form of $f(5)$ is 120.

Termination

One of the most important properties of E is **termination**.

Definition (Terminating)

Given an equational theory $\mathcal{E} = (\Sigma, E \cup A)$, \mathcal{E} is **terminating** iff for any Σ -term t , there does not exist an infinite sequence of rewriting from t by E .

Question

How to decide if an equational theory \mathcal{E} is terminating?

Question

How to decide if an equational theory \mathcal{E} is terminating?

Theorem (Undecidability of termination)

There does not exist such an algorithm to decide whether an arbitrary equational theory \mathcal{E} is terminating or not.

Question

How to decide if an equational theory \mathcal{E} is terminating?

Theorem (Undecidability of termination)

There does not exist such an algorithm to decide whether an arbitrary equational theory \mathcal{E} is terminating or not.

How to prove it?

Back to the quiz

Are the following two equations **terminating**?

$$f(f(x)) = f(g(f(x))) \quad (1)$$

$$g(x) = x \quad (2)$$

Back to the quiz

Are the following two equations **terminating**?

$$f(f(x)) = f(g(f(x))) \quad (1)$$

$$g(x) = x \quad (2)$$

$$f(f(x)) \Rightarrow f(g(f(x))) \Rightarrow f(f(x))$$

Note that: $g(f(x)) = f(x)$ according to the second equation

Back to the quiz

Are the following two equations **terminating**?

$$f(f(x)) = f(g(f(x))) \quad (1)$$

$$g(x) = x \quad (2)$$

$$f(f(x)) \Rightarrow f(g(f(x))) \Rightarrow f(f(x))$$

Note that: $g(f(x)) = f(x)$ according to the second equation

Non-terminating!

Back to the quiz

How about this?

$$0 + x = x \tag{1}$$

$$s(x) + y = s(x + y) \tag{2}$$

Why?

Termination

Example (下面的 TRS 是否是可终止的？)

$$\mathcal{R}_1 = \{f(f(x)) \rightarrow s(s(f(x)))\}$$

Termination

Example (下面的 TRS 是否是可终止的?)

$$\mathcal{R}_1 = \{f(f(x)) \rightarrow s(s(f(x)))\} \quad \checkmark$$

Termination

Example (下面的 TRS 是否是可终止的?)

$$\mathcal{R}_1 = \{f(f(x)) \rightarrow s(s(f(x)))\} \quad \checkmark$$

$$\mathcal{R}_2 = \{f(f(x)) \rightarrow f(g(f(x))), \quad g(x) \rightarrow x\}$$

Termination

Example (下面的 TRS 是否是可终止的?)

$$\mathcal{R}_1 = \{f(f(x)) \rightarrow s(s(f(x)))\} \quad \checkmark$$

$$\mathcal{R}_2 = \{f(f(x)) \rightarrow f(g(f(x))), \quad g(x) \rightarrow x\} \quad \times$$

Termination

Example (下面的 TRS 是否是可终止的 ?)

$$\mathcal{R}_1 = \{f(f(x)) \rightarrow s(s(f(x)))\} \quad \checkmark$$

$$\mathcal{R}_2 = \{f(f(x)) \rightarrow f(g(f(x))), \quad g(x) \rightarrow x\} \quad \times$$

$$\mathcal{R}_3 = \{b(a(x)) \rightarrow a(b(x))\}$$

Termination

Example (下面的 TRS 是否是可终止的 ?)

$$\mathcal{R}_1 = \{f(f(x)) \rightarrow s(s(f(x)))\} \quad \checkmark$$

$$\mathcal{R}_2 = \{f(f(x)) \rightarrow f(g(f(x))), \quad g(x) \rightarrow x\} \quad \times$$

$$\mathcal{R}_3 = \{b(a(x)) \rightarrow a(b(x))\} \quad \checkmark$$

Termination

Example (下面的 TRS 是否是可终止的?)

$$\mathcal{R}_1 = \{f(f(x)) \rightarrow s(s(f(x)))\} \quad \checkmark$$

$$\mathcal{R}_2 = \{f(f(x)) \rightarrow f(g(f(x))), \quad g(x) \rightarrow x\} \quad \times$$

$$\mathcal{R}_3 = \{b(a(x)) \rightarrow a(b(x))\} \quad \checkmark$$

$$\mathcal{R}_4 = \{0 + y \rightarrow y, x + y = y + x, s(x) + y \rightarrow s(x + y)\}$$

Termination

Example (下面的 TRS 是否是可终止的?)

$$\mathcal{R}_1 = \{f(f(x)) \rightarrow s(s(f(x)))\} \quad \checkmark$$

$$\mathcal{R}_2 = \{f(f(x)) \rightarrow f(g(f(x))), \quad g(x) \rightarrow x\} \quad \times$$

$$\mathcal{R}_3 = \{b(a(x)) \rightarrow a(b(x))\} \quad \checkmark$$

$$\mathcal{R}_4 = \{0 + y \rightarrow y, x + y = y + x, s(x) + y \rightarrow s(x + y)\} \quad \times$$

Termination

Example (下面的 TRS 是否是可终止的?)

$$\mathcal{R}_1 = \{f(f(x)) \rightarrow s(s(f(x)))\} \quad \checkmark$$

$$\mathcal{R}_2 = \{f(f(x)) \rightarrow f(g(f(x))), \quad g(x) \rightarrow x\} \quad \times$$

$$\mathcal{R}_3 = \{b(a(x)) \rightarrow a(b(x))\} \quad \checkmark$$

$$\mathcal{R}_4 = \{0 + y \rightarrow y, x + y = y + x, s(x) + y \rightarrow s(x + y)\} \quad \times$$

$$\mathcal{R}_5 = \{0 + y \rightarrow y, s(x) + y \rightarrow s(x + y)\}$$

Termination

Example (下面的 TRS 是否是可终止的?)

$$\mathcal{R}_1 = \{f(f(x)) \rightarrow s(s(f(x)))\} \quad \checkmark$$

$$\mathcal{R}_2 = \{f(f(x)) \rightarrow f(g(f(x))), \quad g(x) \rightarrow x\} \quad \times$$

$$\mathcal{R}_3 = \{b(a(x)) \rightarrow a(b(x))\} \quad \checkmark$$

$$\mathcal{R}_4 = \{0 + y \rightarrow y, x + y = y + x, s(x) + y \rightarrow s(x + y)\} \quad \times$$

$$\mathcal{R}_5 = \{0 + y \rightarrow y, s(x) + y \rightarrow s(x + y)\} \quad \checkmark$$

Think about the following problem

Is it $\mathcal{E} = (\Sigma, E)$ terminating or not, where:?

$$\Sigma = \{+, -, \times, \div, \alpha, \beta, 0, 1, \phi\}$$

$$\begin{aligned} E = & \{\phi(x + y) \rightarrow \phi(x) + \phi(y), \\ & \phi(x - y) \rightarrow \phi(x) - \phi(y), \\ & \phi(x \times y) \rightarrow (\phi(x) \times y) + (x \times \phi(y)), \\ & \phi(x \div y) \rightarrow ((\phi(x) \times y) - (x \times \phi(y))) \div (y \times y), \\ & \phi(\alpha) \rightarrow 1, \\ & \phi(\beta) \rightarrow 0\}. \end{aligned}$$

代数 (Algebra)

什么叫代数？

代数 (Algebra)

什么叫代数？**代数是一个集合及定义在该集合上的一组运算，如 $(\mathbb{N}, +)$**

$\mathcal{F} : \text{Signature}$

Definition

\mathcal{F} -algebra $\mathcal{A} = (A, \{f_{\mathcal{A}}\}_{f \in \mathcal{F}})$ consists of

- carrier A
- interpretation: $f_{\mathcal{A}} : \overbrace{A \times \dots \times A}^n \rightarrow A$ if $f^{(n)} \in \mathcal{F}$

Definition

- assignment: $\alpha : \mathcal{V} \rightarrow A$

- interpretation function: $[\alpha]_{\mathcal{A}} : \mathcal{T}(\mathcal{F}, \mathcal{V}) \rightarrow A$

$$[\alpha]_{\mathcal{A}}(t) = \begin{cases} \alpha(t) & \text{if } t \in \mathcal{V} \\ f_{\mathcal{A}}([\alpha]_{\mathcal{A}}(t_1), \dots, [\alpha]_{\mathcal{A}}(t_n)) & \text{if } t = f(t_1, \dots, t_n) \end{cases}$$

例子

Example

Let $\mathcal{F} = \{0, s, \oplus\}$ and $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}\})$

$$0_{\mathcal{A}} = 1 \quad s_{\mathcal{A}}(x) = x + 1 \quad \oplus_{\mathcal{A}}(x, y) = x + x + y$$

For assignment α with $\alpha(x) = 10$ and $\alpha(y) = 2$, we have

$$[\alpha]_{\mathcal{A}}(s(x) \oplus y) =$$

例子

Example

Let $\mathcal{F} = \{0, s, \oplus\}$ and $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}\})$

$$0_{\mathcal{A}} = 1 \quad s_{\mathcal{A}}(x) = x + 1 \quad \oplus_{\mathcal{A}}(x, y) = x + x + y$$

For assignment α with $\alpha(x) = 10$ and $\alpha(y) = 2$, we have

$$[\alpha]_{\mathcal{A}}(s(x) \oplus y) = [\alpha]_{\mathcal{A}}(s(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y)$$

例子

Example

Let $\mathcal{F} = \{0, s, \oplus\}$ and $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}\})$

$$0_{\mathcal{A}} = 1 \quad s_{\mathcal{A}}(x) = x + 1 \quad \oplus_{\mathcal{A}}(x, y) = x + x + y$$

For assignment α with $\alpha(x) = 10$ and $\alpha(y) = 2$, we have

$$\begin{aligned} [\alpha]_{\mathcal{A}}(s(x) \oplus y) &= [\alpha]_{\mathcal{A}}(s(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \end{aligned}$$

例子

Example

Let $\mathcal{F} = \{0, s, \oplus\}$ and $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}\})$

$$0_{\mathcal{A}} = 1 \quad s_{\mathcal{A}}(x) = x + 1 \quad \oplus_{\mathcal{A}}(x, y) = x + x + y$$

For assignment α with $\alpha(x) = 10$ and $\alpha(y) = 2$, we have

$$\begin{aligned} [\alpha]_{\mathcal{A}}(s(x) \oplus y) &= [\alpha]_{\mathcal{A}}(s(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= s_{\mathcal{A}}(10) \oplus_{\mathcal{A}} 2 \end{aligned}$$

例子

Example

Let $\mathcal{F} = \{0, s, \oplus\}$ and $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}\})$

$$0_{\mathcal{A}} = 1 \quad s_{\mathcal{A}}(x) = x + 1 \quad \oplus_{\mathcal{A}}(x, y) = x + x + y$$

For assignment α with $\alpha(x) = 10$ and $\alpha(y) = 2$, we have

$$\begin{aligned} [\alpha]_{\mathcal{A}}(s(x) \oplus y) &= [\alpha]_{\mathcal{A}}(s(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= s_{\mathcal{A}}(10) \oplus_{\mathcal{A}} 2 \\ &= 11 \oplus_{\mathcal{A}} 2 \end{aligned}$$

例子

Example

Let $\mathcal{F} = \{0, s, \oplus\}$ and $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}\})$

$$0_{\mathcal{A}} = 1 \quad s_{\mathcal{A}}(x) = x + 1 \quad \oplus_{\mathcal{A}}(x, y) = x + x + y$$

For assignment α with $\alpha(x) = 10$ and $\alpha(y) = 2$, we have

$$\begin{aligned} [\alpha]_{\mathcal{A}}(s(x) \oplus y) &= [\alpha]_{\mathcal{A}}(s(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= s_{\mathcal{A}}(10) \oplus_{\mathcal{A}} 2 \\ &= 11 \oplus_{\mathcal{A}} 2 \\ &= 24 \end{aligned}$$

例子

Example

Let $\mathcal{F} = \{0, s, \oplus\}$ and $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}\})$

$$0_{\mathcal{A}} = 1 \quad s_{\mathcal{A}}(x) = x + 1 \quad \oplus_{\mathcal{A}}(x, y) = x + x + y$$

For assignment α with $\alpha(x) = 10$ and $\alpha(y) = 2$, we have

$$\begin{aligned} [\alpha]_{\mathcal{A}}(s(x) \oplus y) &= [\alpha]_{\mathcal{A}}(s(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= s_{\mathcal{A}}(10) \oplus_{\mathcal{A}} 2 \\ &= 11 \oplus_{\mathcal{A}} 2 \\ &= 24 \end{aligned}$$

$$[\alpha]_{\mathcal{A}}(s(x \oplus y)) = s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x \oplus y)) = 23$$

证明

证明

Let $\mathcal{F} = \{0, s, \oplus\}$ and $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}\})$

$$0_{\mathcal{A}} = 1 \quad s_{\mathcal{A}}(x) = x + 1 \quad +_{\mathcal{A}}(x, y) = x + x + y$$

对任意的 assignment $\alpha : \mathcal{V} \rightarrow \mathbb{N}$, 下面的不等式是否成立:

$$[\alpha]_{\mathcal{A}}(s(x) \oplus y) > [\alpha]_{\mathcal{A}}(s(x \oplus y))$$

证明

证明

Let $\mathcal{F} = \{0, s, \oplus\}$ and $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}\})$

$$0_{\mathcal{A}} = 1 \quad s_{\mathcal{A}}(x) = x + 1 \quad +_{\mathcal{A}}(x, y) = x + x + y$$

对任意的 assignment $\alpha : \mathcal{V} \rightarrow \mathbb{N}$, 下面的不等式是否成立:

$$[\alpha]_{\mathcal{A}}(s(x) \oplus y) > [\alpha]_{\mathcal{A}}(s(x \oplus y))$$

Proof.

$$\begin{aligned} \text{左边} &= [\alpha]_{\mathcal{A}}(s(x) \oplus y) = [\alpha]_{\mathcal{A}}(s(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) = s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= ([\alpha]_{\mathcal{A}}(x) + 1) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) = ([\alpha]_{\mathcal{A}}(x) + 1) + ([\alpha]_{\mathcal{A}}(x) + 1) + [\alpha]_{\mathcal{A}}(y) \\ &= [\alpha]_{\mathcal{A}}(x) + [\alpha]_{\mathcal{A}}(x) + [\alpha]_{\mathcal{A}}(y) + 2 \end{aligned}$$

证明

证明

Let $\mathcal{F} = \{0, s, \oplus\}$ and $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}\})$

$$0_{\mathcal{A}} = 1 \quad s_{\mathcal{A}}(x) = x + 1 \quad +_{\mathcal{A}}(x, y) = x + x + y$$

对任意的 assignment $\alpha : \mathcal{V} \rightarrow \mathbb{N}$, 下面的不等式是否成立:

$$[\alpha]_{\mathcal{A}}(s(x) \oplus y) > [\alpha]_{\mathcal{A}}(s(x \oplus y))$$

Proof.

$$\begin{aligned}\text{左边} &= [\alpha]_{\mathcal{A}}(s(x) \oplus y) = [\alpha]_{\mathcal{A}}(s(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) = s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x)) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) \\ &= ([\alpha]_{\mathcal{A}}(x) + 1) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) = ([\alpha]_{\mathcal{A}}(x) + 1) + ([\alpha]_{\mathcal{A}}(x) + 1) + [\alpha]_{\mathcal{A}}(y) \\ &= [\alpha]_{\mathcal{A}}(x) + [\alpha]_{\mathcal{A}}(x) + [\alpha]_{\mathcal{A}}(y) + 2\end{aligned}$$

$$\begin{aligned}\text{右边} &= [\alpha]_{\mathcal{A}}(s(x \oplus y)) = s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x \oplus y)) = s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y)) \\ &= s_{\mathcal{A}}([\alpha]_{\mathcal{A}}(x) + [\alpha]_{\mathcal{A}}(x) + [\alpha]_{\mathcal{A}}(y)) = [\alpha]_{\mathcal{A}}(x) + [\alpha]_{\mathcal{A}}(x) + [\alpha]_{\mathcal{A}}(y) + 1\end{aligned}$$

□

判定是否可终止的充分条件

Definition (单调 (monotone))

Algebra $\mathcal{A} = (\mathbb{N}, \{f_{\mathcal{A}}\}_{f \in \mathcal{F}})$ is monotone if

$$f(a_1, \dots, a_i, \dots, a_n) > f(a_1, \dots, b_i, \dots, a_n)$$

for all $f^{(n)} \in \mathcal{F}$ and all $a_1, \dots, a_n, b_i \in \mathbb{N}$, $a_i > b_i$

Theorem

A TRS \mathcal{R} is terminating if there is monotone algebra on \mathbb{N} such that

$$[\alpha]_{\mathcal{A}}(\ell) > [\alpha]_{\mathcal{A}}(r)$$

for all rules $\ell \rightarrow r \in \mathcal{R}$ and assignment α .

例子

Example

TRS \mathcal{R} :

$$\begin{array}{ll} 0 \oplus y \rightarrow y & 0 \otimes y \rightarrow 0 \\ s(x) \oplus y \rightarrow s(x \oplus y) & s(x) \otimes y \rightarrow y \oplus (x \otimes y) \end{array}$$

定义一个在 \mathbb{N} 上的单调代数 $\mathcal{A} = (\mathbb{N}, \{0_{\mathcal{A}}, s_{\mathcal{A}}, \oplus_{\mathcal{A}}, \otimes_{\mathcal{A}}\})$:

$$\begin{array}{ll} 0_{\mathcal{A}} = 1 & x \oplus_{\mathcal{A}} y = 2x + y \\ s_{\mathcal{A}}(x) = x + 1 & x \otimes_{\mathcal{A}} y = 2xy + x + y + 1 \end{array}$$

证明对任意 $\ell \rightarrow r \in \mathcal{R}$ 和 assignment α , 满足 $[\alpha]_{\mathcal{A}}(\ell) > [\alpha]_{\mathcal{A}}(r)$:

- 1 $[\alpha]_{\mathcal{A}}(0 \oplus y) = [\alpha]_{\mathcal{A}}(0) \oplus_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) = 2 + [\alpha]_{\mathcal{A}}(y) > [\alpha]_{\mathcal{A}}(y)$
- 2 $[\alpha]_{\mathcal{A}}(0 \times y) = [\alpha]_{\mathcal{A}}(0) \otimes_{\mathcal{A}} [\alpha]_{\mathcal{A}}(y) = 3[\alpha]_{\mathcal{A}}(y) + 2 > [\alpha]_{\mathcal{A}}(y)$
- 3 ...
- 4 ...