

CDM

Equational Reasoning

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner

Battleplan

- Example: Associativity
- Equational Logic
- Equational Reasoning
- Rewrite Systems
- Unification

Example: Associativity

Beyond Propositional Logic

SAT Solvers have become a standard computational tool and are surprisingly powerful, given the computational hardness results associated with the satisfiability.

But how about situations that can not be described by (even a large number of) propositional variables and logical connectives? For example, the assertion that every non-zero element in a field has an inverse:

$$x \neq 0 \rightarrow i(x) \cdot x = 1$$

We could move to full first order logic to deal with such assertions but there is an immediate step that is of independent interest: *equational logic*. Equational logic is expressive enough to describe interesting situations yet simple enough to admit reasonably efficient algorithms.

Cheap Example: Associativity

Recall that a binary operation is *associative* if it does not matter where we place the parentheses in an expression: the final result of an evaluation is always the same. Associative operations abound in algebra, addition and multiplication of various kinds of numbers (naturals, integers, rationals, reals, complexes) or matrices are always associative. Typical associative operations in CS are concatenation of words and join of lists.

The usual counterexample is exponentiation: $2^{3^2} \neq (2^3)^2$.

Pairing operations when implemented by some kind of record data structure also fail to be associative: $\langle a, \langle b, c \rangle \rangle$ is usually not the same as $\langle \langle a, b \rangle, c \rangle$ (though they are in some sense “equivalent”).

The Associative Law

Associativity is usually expressed in terms of an equation:

$$(x * y) * z = x * (y * z)$$

The understanding here is that x , y and z are free variables that can be replaced by arbitrary objects in the domain in question. On a more syntactical level, they could be replaced by arbitrary expressions.

At first glance, this associativity axiom may seem to be too weak an assertion: it only guarantees that in expressions with two occurrences of operation $*$ the placement of the parentheses does not matter. But how about, say,

$$((a * b) * c) * d \quad \text{versus} \quad a * (b * (c * d))$$

The last two expressions are in fact equal by associativity. But exactly how does this follow? More importantly, can we generate such a proof automatically?

An Equational Proof

Here is a proof that uses only the associativity assumption and some equational reasoning. We need to determine which terms have to be assigned to the variables in the axiom to get the desired effect.

$$\begin{array}{l}
 ((a * b) * c) * d \\
 \downarrow \quad a * b/x, c/y, d/z \\
 (a * b) * (c * d) \\
 \downarrow \quad a/x, b/y, c * d/z \\
 a * (b * (c * d))
 \end{array}$$

Here $a * b/x$ means: substitute $a * b$ for x and so on.

Hence the application of the given associativity equation requires some sort of pattern matching: we have to determine how to bind the variables to terms.

Moving Parens Around

Emboldened by first success, let's try something more ambitious.

Claim 1. *Associativity implies*

$$(\dots (x_1 * x_2) * \dots * x_{n-1}) * x_n = x_1 * (x_2 * (\dots (x_{n-1} * x_n) \dots))$$

Proof.

Note that this is really not just one claim but infinitely many: one for each value of $n \geq 0$. Unsurprisingly, we will use induction along the way.

First we establish a little auxiliary result: we can “pull out” the first term.

$$(\dots ((\mathbf{a} * x_1) * x_2) * \dots * x_{n-1}) * x_n = \mathbf{a} * ((\dots (x_1 * x_2) * \dots * x_{n-1}) * x_n)$$

A Proof

We prove the auxiliary result by induction on n .

$n = 0$ is easy, so assume the claim holds for $n - 1 \geq 0$.

$$((\dots ((\mathbf{a} * x_1) * x_2) * \dots * x_{n-1}) * x_n =$$

by induction hypothesis

$$(\mathbf{a} * (\dots (x_1 * x_2) * \dots * x_{n-1})) * x_n =$$

by associativity

$$\mathbf{a} * ((\dots (x_1 * x_2) * \dots * x_{n-1}) * x_n)$$

Main Argument

Case $n = 3$ is just the associativity axiom. So assume we have the claim for $n \geq 3$. Then

$$(\dots((\mathbf{a} * x_1) * x_2) * \dots * x_{n-1}) * x_n =$$

by auxiliary claim

$$\mathbf{a} * ((\dots(x_1 * x_2) * \dots * x_{n-1}) * x_n) =$$

by IH

$$\mathbf{a} * (x_1 * (x_2 * (\dots(x_{n-1} * x_n) \dots)))$$

□

Some Comments on the Proof

It is important to note that each of the formulae

$$\varphi_n : (\dots (x_1 * x_2) * \dots * x_{n-1}) * x_n = x_1 * (x_2 * (\dots (x_{n-1} * x_n) \dots))$$

can be proven from the associativity axiom, using a separate proof for each n .

We are not currently interested in a proof for $\forall n \varphi_n$ or some such, this would require a much more powerful system.

And don't even think about forming an infinite conjunction

$$\varphi_0 \wedge \varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n \wedge \dots$$

A Challenge

So we have proven that the associativity axiom has the consequence

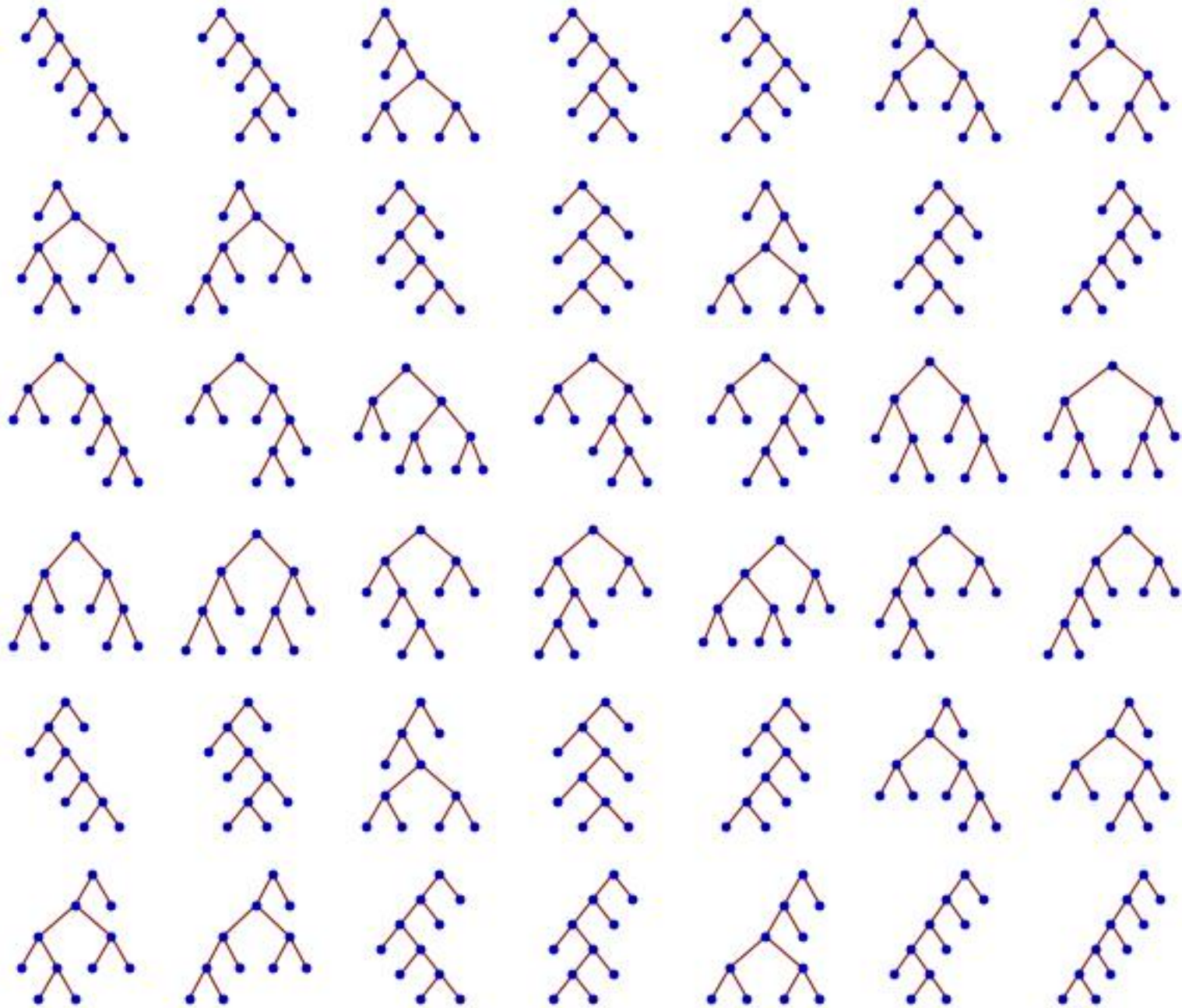
$$(\dots (x_1 * x_2) * \dots * x_{n-1}) * x_n = x_1 * (x_2 * (\dots (x_{n-1} * x_n) \dots))$$

Of course, this is just one way to parenthesize these expressions, a more ambitious project would be to show that a similar equation can be established for all possible parenthesizations.

Which brings up the question what exactly is meant by “all possible parenthesizations”. A good model is the parse tree of an expression: internal nodes denote an occurrence of $*$ and the leaves are labeled by the arguments: if we read off the frontier of the tree from left to right we get x_1, x_2, \dots, x_n .

Note that these trees are full binary trees. We need to show that the associativity axiom, when construed as an equation on full binary trees, allows us to transform all full binary trees with n leaves into one another.

Frivolous Picture



A Non-Theorem

Here is a formula that cannot be derived from associativity:

$$x * y = y * x$$

Operations that satisfy this property are called *commutative* or *Abelian*.

Note that in order to show that this formula is not derivable from the associativity axiom it is enough to find just one structure in which associativity holds but this formula is false: the single counterexample shows that φ is not a semantic consequence of the associativity axiom, and so it cannot be derivable either.

Non-commutative associative operations are plentiful: take all words over a two-letter alphabet, say, $\{a, b\}$, with concatenation. Clearly $ab \neq ba$ so commutativity fails.

Matrix multiplication is another classical example.

Who Cares?

It was recognized in the 1970's that data structures should be modeled independent from any concrete implementation (*abstract data types*).

- Algebraic specification of abstract datatypes via signatures and axiomatic descriptions of the primitive operations:
 - Declaration of used types.
 - Declaration of primitive operations.
 - Axioms specifying behaviour of operations.
- The description contains virtually no information about implementation.

The lack of implementation helps to simplify the description and provides a clear interface – which is crucial for reuse and for any attempt at correctness arguments.

ADT List

Here is an ADT type description of a list data structure based on the primitive operation of prepend. Here is the vocabulary:

a, b, c, \dots	atoms
nil	empty list
x, y, z, \dots	lists
$\text{prep}(a, x)$	prepend atom a to list x

Specifications: prepending an atom to nil produces the atom, after a prepend, the list is not empty and there is a unique way to decompose a list produced by repeated prepends:

$$\text{prep}(a, \text{nil}) = a$$

$$\text{prep}(a, x) \neq \text{nil}$$

$$\text{prep}(a, x) = \text{prep}(b, y) \rightarrow a = b \wedge x = y$$

These are the given elementary properties that an implementation is supposed to satisfy.

Joins

Given prepend, one can define other operations such as the join of two lists by induction.

$$\text{join}(\text{nil}, y) = y$$

$$\text{join}(\text{prep}(a, x), y) = \text{prep}(a, \text{join}(x, y))$$

This definition will then guarantee certain properties such as associativity.

Note that induction is at work in this definition: we are really defining $\text{join}(x, y)$ by induction on the length of x .

Reversal

Now suppose you want to add another operation, reversal.

$$\text{rev}(\text{nil}) = \text{nil}$$

$$\text{rev}(\text{prep}(a, x)) = \text{join}(\text{rev}(x), a)$$

The claim is that given these specifications it follows that

$$\text{rev}(\text{join}(x, y)) = \text{join}(\text{rev}(y), \text{rev}(x)).$$

To see this first show that $\text{join}(x, \text{nil}) = x$ by induction on x and then the claim, again by induction on x . Apart from induction, the whole argument is equational.

Exercises

- Exercise 1.** *Show that join is an associative operation.*
- Exercise 2.** *Prove the join-reversal property.*
- Exercise 3.** *How would you go about adding a predicate empty to this system?*
- Exercise 4.** *How would you go about adding operations head and tail to this system?*
- Exercise 5.** *Define a similar definition for binary trees.*

Equational Logic

Syntax for Equations

With a view towards implementation, let's take a closer look at the syntactic aspects of equations (also called identities in this context).

We need a simple language to construct terms.

- A collection Var of variables, written $x, y, z \dots$,
- A collection Σ of function symbols, written $f, g, h \dots$
- A map $\text{ar} : \Sigma \rightarrow \mathbb{N}$ that provides an arity for each function symbol in Σ .

$\langle \Sigma, \text{ar} \rangle$ is called a *graded alphabet*.

To lighten notation we use subscripts, superscripts and the like.

Also, function symbols of arity 0 are really constants and we write $a, b, c \dots$ for them.

Binary function symbols are often written in infix notation as in $x \oplus y$ rather than the pedantic $\oplus(x, y)$.

Terms

Definition 1. We define the set of terms $\mathcal{T} = \mathcal{T}(\Sigma) = \mathcal{T}(\text{Var}, \Sigma)$ over Var and Σ by induction:

- Every variable $x \in \text{Var}$ is a term, and
- given terms t_1, \dots, t_n and a function symbol $f \in \Sigma$ of arity n , $f(t_1, \dots, t_n)$ is a term.

An **equation** or **identity** is an expression

$$s \approx t$$

where s and t are terms.

Note that this is just syntactic sugar, we might as well have defined an identity to be a pair (s, t) .

Also, we have deliberately chosen to write \approx rather than the customary equality sign $=$ to make clear that we are defining a special class of syntactic objects. Later we will be sloppy and write $s = t$.

Examples

Example 1. *The usual associative law has $\Sigma = \{*\}$ where $\text{ar}(*) = 2$:*

$$x * (y * z) \approx (x * y) * z \quad (\text{assoc})$$

Example 2. *To write equations in elementary arithmetic we could use 4 function symbols $\Sigma = \{\oplus, \otimes, S, \mathbf{0}, \mathbf{1}\}$ where $\text{ar}(\oplus) = \text{ar}(\otimes) = 2$, where $\text{ar}(S) = 1$ and $\text{ar}(\mathbf{0}) = \text{ar}(\mathbf{1}) = 0$.*

Typical examples of equations are

$$x \oplus y \approx y \oplus x$$

$$x \oplus \mathbf{0} \approx x$$

$$x \oplus S(y) \approx S(x \oplus y)$$

Semantics

So far, an equation is a purely syntactic object. In order to attach meaning to these expressions we need to interpret them over a concrete structure, a domain of discourse.

Suppose $\Sigma = \{f_1, \dots, f_k\}$. Then a suitable structure has the form

$$\mathcal{A} = \langle A, F_1, F_2, \dots, F_k \rangle$$

where $F_i : A^{\text{ar}(f_i)} \rightarrow A$: F_i is an actual function of the right arity that is used to interpret the function symbol f_i .

Again: f_i is a purely syntactical object, and F_i is its meaning in the real world (over a particular structure \mathcal{A}).

Valuations

Definition 2. A **valuation** or **assignment** is a map that assigns elements in a structure to variables:

$$\sigma : \text{Var} \rightarrow A$$

By a straightforward induction we can extend a valuation to all terms (rather than just variables):

$$\sigma(f(t_1, \dots, t_n)) = F(\sigma(t_1), \dots, \sigma(t_n)).$$

This is just the standard process of evaluation of an expression, given values for all variables. f here is just a syntactic object, a symbol, whereas F is the actual function (the implementation, perhaps even a piece of executable code).

Validity

Now consider an equation $s \approx t$ and a valuation σ over some structure \mathcal{A} .

Definition 3. $s \approx t$ is valid over \mathcal{A} with respect to σ if $\sigma(s) = \sigma(t)$. We also say that \mathcal{A} is a **model** of $s \approx t$. $s \approx t$ is valid over \mathcal{A} if it is valid for all valuations.

We use the same terminology for a set E of equations: E is valid over \mathcal{A} if all the equations in E are so valid.

Notation:

$$\mathcal{A} \models_{\sigma} s \approx t$$

$$\mathcal{A} \models s \approx t$$

For a set of equations E we require that all the equations in E hold over \mathcal{A} :

$$\mathcal{A} \models E$$

Example: Group Axioms

The point is that very simple equations can have startlingly complicated models.

For example, to pin down groups we choose a graded alphabet $\Sigma = (*, {}^{-1}, 1)$ with arities $(2, 1, 0)$. There are just five equations:

$$x * (y * z) \approx (x * y) * z$$

$$x * x^{-1} \approx x^{-1} * x \approx 1$$

$$x * 1 \approx 1 * x \approx x$$

This is not the only choice, one could also use $\Sigma = (*, 1)$ or even $\Sigma = (*)$ but then the basic definitions require quantification.

Models

A structure suitable for this alphabet must be of the form

$$\mathcal{A} = \langle A; f, g, c \rangle$$

where $f : A \times A \rightarrow A$, $g : A \rightarrow A$ and $c \in A$.

One is often a bit sloppy and uses more suggestive notation such as

$$\mathcal{A} = \langle A; *, {}^{-1}, 1 \rangle \quad \mathcal{A} = \langle A; +, -, 0 \rangle$$

and the like.

Group Examples

Typical examples:

The additive group of integers:

A integers
 $*$ addition
 -1 unary minus
 1 zero

The multiplicative group of positive reals:

A positive reals
 $*$ multiplication
 -1 inverse
 1 one

Groups

Many important monoids satisfy another property:

$$\forall x \exists y (x * y \approx e \wedge y * x \approx e) \tag{1}$$

Definition 4. *The element y above is called an inverse of x . A monoid with this additional property is a group.*

Example 3. *The monoid from above,*

<i>*</i>	<i>e</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>f</i>
<i>e</i>	<i>e</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>f</i>
<i>a</i>	<i>a</i>	<i>c</i>	<i>f</i>	<i>e</i>	<i>b</i>	<i>d</i>
<i>b</i>	<i>b</i>	<i>f</i>	<i>a</i>	<i>d</i>	<i>e</i>	<i>c</i>
<i>c</i>	<i>c</i>	<i>e</i>	<i>d</i>	<i>a</i>	<i>f</i>	<i>b</i>
<i>d</i>	<i>d</i>	<i>b</i>	<i>e</i>	<i>f</i>	<i>c</i>	<i>a</i>
<i>f</i>	<i>f</i>	<i>d</i>	<i>c</i>	<i>b</i>	<i>a</i>	<i>e</i>

is a group.

Some Very Important Groups

$$\langle \mathbb{Z}; +, 0 \rangle$$

$$\langle \mathbb{Q}^+; \cdot, 1 \rangle$$

$$\langle \mathbb{R}^+; \cdot, 1 \rangle$$

$$\mathbb{Z}_n = \langle \{0, 1, \dots, n-1\}; \oplus, 0 \rangle$$

$$\mathbb{Z}_p^* = \langle \{1, \dots, p-1\}; \otimes, 1 \rangle$$

Here $x \oplus y$ is $x + y \bmod n$. Unsigned ints in C behave this way for $n = 2^{32}$.

$x \otimes y$ is $x \cdot y \bmod p$ and p is required to be a prime (otherwise we have a monoid, but not a group).

A Theorem in Group Theory

We already have a set Γ of 3 axioms for group theory. Note that we use signature $(2, 0)$ so that there is no function symbol for the inverse. But we can prove the following.

Claim. *The inverse element is always unique:*

$$\text{inv}(x, y) \wedge \text{inv}(x, z) \rightarrow z \approx y$$

Here we have written $\text{inv}(x, y)$ as abbreviation for $x * y \approx e \wedge y * x \approx e$.

Since we have not explicitly stated a proof system for FOL we cannot give a formal proof, but it should be clear that any reasonable system will support equational reasoning: it is admissible to replace terms by equal terms.

Proof

Here is a purely equational proof that could easily be handled in any formal system.

Proof. Assuming $\text{inv}(x, y) \wedge \text{inv}(x, z)$ we have

$$\begin{aligned} z &\approx e * z \\ &\approx (y * x) * z \\ &\approx y * (x * z) \\ &\approx y * e \\ &\approx y \end{aligned}$$

□

Exercise 6. *Determine precisely where the group axioms are used in this argument.*

Inverse and Products

Because of the claim we are now justified in introducing a special function symbol for the inverse. As usual, this is written in exponential notation: x^{-1} for the uniquely determined element y such that $\text{inv}(x, y)$.

So we define as an abbreviation

$$x^{-1} \approx y \iff \text{inv}(x, y)$$

We can then prove the following lemma which describes the interaction between $^{-1}$ and multiplication.

Lemma 1.

$$(x * y)^{-1} \approx y^{-1} * x^{-1}$$

Exercise 7. *Prove the last lemma making sure you only use the group axioms and the abbreviation for the inverse operation.*

Abelian Groups

All the examples of groups we have seen so far are commutative.

Could it be true that commutativity follows somehow from the group axioms?

No, there are lots of non-commutative groups. Here is a table of the smallest non-commutative group. It has 6 elements.

<i>*</i>	<i>e</i>	<i>r</i>	<i>s</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>e</i>	<i>e</i>	<i>r</i>	<i>s</i>	<i>a</i>	<i>b</i>	<i>c</i>
<i>r</i>	<i>r</i>	<i>s</i>	<i>e</i>	<i>b</i>	<i>c</i>	<i>a</i>
<i>s</i>	<i>s</i>	<i>e</i>	<i>r</i>	<i>c</i>	<i>a</i>	<i>b</i>
<i>a</i>	<i>a</i>	<i>c</i>	<i>b</i>	<i>e</i>	<i>s</i>	<i>r</i>
<i>b</i>	<i>b</i>	<i>a</i>	<i>c</i>	<i>r</i>	<i>e</i>	<i>s</i>
<i>c</i>	<i>c</i>	<i>b</i>	<i>a</i>	<i>s</i>	<i>r</i>	<i>e</i>

Checking the group properties requires work, as usual, but non-commutativity is easy to see. Note the 3 by 3 square in the upper left hand corner: $\{e, r, s\}$ is a *commutative subgroup*.

Sporadic Groups

Though the equations are nearly trivial, just to understand their finite models took a huge amount of effort (classification of finite simple groups, completed 1985, some 15,000 pages of proofs).

The theorem says in essence that all finite groups can be decomposed into just 5 basic types. Two of these basic groups are very straightforward, e.g., cyclic groups of prime order or alternating groups, two are a bit more messy (certain Lie-type groups) but the 5th group is utterly bizarre: it consists of 26 rather strange groups.

The largest one is called the *monster*. It is a group of rotations, albeit in 196883-dimensional space. Its size is

808017424794512875886459904961710757005754368000000000

How on earth does this number come out of the group axioms?

A Trivial Model

On the other hand, any purely equational set of axioms always has a model.

Proposition 1. *Every system of equations has a model.*

Proof.

A trivial model can be constructed by choosing $A = \{\bullet\}$ and setting all functions to be constant with value \bullet .

□

Of course, this trivial model is utterly useless, we need a carrier set of size at least 2 for an equation to be interesting. Things change drastically if we allow inequalities or implications between equations. For example, the field axioms force the existence of at least two elements zero and one.

Exercise 8. *Meditate deeply on the nature of \bullet .*

Semantic Consequence

As we have seen for the associative law, it may well happen that the validity of one equation in a structure entails the validity of others. More generally, suppose E is a system of equations and e a single equation.

Definition 5. e is a semantic consequence of E if for all structures \mathcal{A} such that $\mathcal{A} \models E$ we also have $\mathcal{A} \models e$.

Notation:

$$E \models e$$

Example 4. Equation $x * (y * (z * u)) \approx ((x * y) * z) * u$ is a semantic consequence of $x * (y * z) \approx (x * y) * z$.

But $x * y = y * x$ is not a consequence of associativity.

Establishing Properties

In the context of the list ADT it suffices to establish compliance with certain basic specifications: more complicated properties of derived operations then follow.

For example, given prep as the only primitive, we can use recursion to definition join and rev.

Given these derived operations, we can use induction and equational reasoning to prove that the join-reversal lemma holds.

Here are some more examples that avoid recursion but are based on a fancier algebraic structure.

Laws of Boolean Algebra

A Boolean algebra is a structure $\mathcal{B} = \langle B, +, \cdot, \bar{}, 0, 1 \rangle$ where the following system of equations (BA) is valid:

$$x + (y + z) = (x + y) + z$$

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z$$

$$x + y = y + x$$

$$x \cdot y = y \cdot x$$

$$x + 0 = x$$

$$x \cdot 1 = x$$

$$x + (y \cdot z) = (x + y) \cdot (x + z)$$

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$$

$$x + \bar{x} = 1$$

$$x \cdot \bar{x} = 0$$

Note that we are in full fudge-mode now, no more \approx , \oplus and the like. But be aware that (BA) is just a collection of formal expressions.

More interestingly, note the duality between plus and times in (BA).

Two Standard Models

The trivial model is useless, it is important to check if there are interesting models. Here are two standard models for (BA) .

- Truth values

The Boolean values “true” and “false” together with disjunction, conjunction and negation:

$$\langle \{ff, tt\}, \vee, \wedge, \neg, ff, tt \rangle$$

- Power sets

The power set of any set, together with union, intersection and complement:

$$\langle \text{pow}(A), \cup, \cap, \neg, \emptyset, A \rangle$$

Exercise 9. *Verify that these structures are indeed Boolean algebras.*

Two More Models

- Finite/cofinite Sets

Let $\text{pow}_f(A)$ be the set of all subsets of A that are either finite or cofinite (the complement is finite).

$$\langle \text{pow}_f(\mathbb{N}), \cup, \cap, \bar{}, \emptyset, \mathbb{N} \rangle$$

- Divisor Lattices

For any positive natural number n let $\text{Div}(n)$ be the set of all divisors of n . Then

$$\langle \text{Div}(n), \text{lcm}, \text{gcd}, n/x, 1, n \rangle$$

is a Boolean algebra provided that n is square-free.

Exercise 10. *Show that the divisor lattice is a Boolean algebra if, and only if, n is square-free. Do these structures look familiar?*

Other Properties

(BA) has many interesting semantic consequences. Some particularly important ones are:

$$x + x = x$$

$$x \cdot x = x$$

$$x + x \cdot y = x$$

$$x \cdot (x + y) = x$$

$$x + 1 = 1$$

$$x \cdot 0 = 0$$

$$\overline{x + y} = \bar{x} \cdot \bar{y}$$

$$\overline{x \cdot y} = \bar{x} + \bar{y}$$

$$\overline{\bar{x}} = x$$

It is easy to see that these all hold in the models above, but the point is that they hold in all models of (BA) . But showing that turns out to be harder than you might think.

Exercise 11. *Verify that these equations hold in the two standard models.*

Proofs?

So how do we go about proving that the Boolean laws have these other equations as consequences?

We cannot argue in terms of models, since the class of all models is huge and we cannot hope to check each and every one of them.

Instead, we have to rely on reasoning about equations. For example, if we know that $s = t$ and $t = u$ hold in a structure, then we can conclude that $s = u$ also holds.

Likewise, we are allowed to substitute arbitrary terms for variables, very much in the way we dealt with associativity above.

The crucial point is that all our manipulations must preserve syntactic consequence. We'll make this more precise in a minute, first some examples.

Sample Proofs

Claim 2. $(BA) \models x + x = x$

Proof.

$$\begin{aligned}x + x &= (x + x) \cdot 1 \\ &= (x + x) \cdot (x + \bar{x}) \\ &= x + x \cdot \bar{x} \\ &= x + 0 \\ &= x\end{aligned}$$

□

Note that the third step uses distributivity of plus over times in the “opposite” direction.

More Proofs

Claim 3. $(BA) \models 1 + x = 1$

Proof.

$$1 + x = (x + \bar{x}) + x = (x + x) + \bar{x} = x + \bar{x} = 1$$

□

Claim 4. $(BA) \models x + xy = x$

Proof.

$$x + xy = x \cdot (1 + y) = x \cdot 1 = x$$

□

Equational Reasoning

Equational Reasoning

Let us try to pin down how these arguments really work: we repeatedly substitute equal terms for equal terms and use the obvious properties of equality.

The given equations in (BA) and the basic properties of equality are the only source for the equalities that can be used in the argument.

As a consequence, the argument is easy to check for correctness: we just have to identify which axiom and what substitution was used.

In an annotated proof these would also be given explicitly so the correctness check comes down to pattern matching.

Alas, the real problem is not to check the proof for correctness but to find it in the first place.

Equational Reasoning, II

Following Gödel's approach, we can define an equational proof of e (over some system E) to be a sequence of equations

$$e_1, e_2, \dots, e_{n-1}, e_n = e$$

where each e_i is either in E , is a trivial identity $s \approx s$ or can be derived from a previous equation e_j , $j < i$ by a simple rule to be spelled out below. For example, we may switch the two sides of an equation.

There is one small exception: for transitivity of equality we need two premises.

Also, a better representation of an equational proof (at least for humans) is a tree with e as the root. A node is the consequence of its children.

Rules: Equality and Axioms

First we need rules that express the fact that equality is reflexive, symmetric and transitive.

$$\frac{-}{s \approx s} \quad \frac{s \approx t}{t \approx s} \quad \frac{s \approx t \quad t \approx u}{s \approx u}$$

The first rule has no premise, the second one, and the last two.

Also, given a system of equations E as assumptions, we are allowed to use any equation $s \approx t \in E$:

$$\frac{-}{s \approx t}$$

Note that these rules are not quite enough: think about the associativity example from above.

Rules: Substitution

Two rules relate to substitutions (instantiation and congruence):

$$\frac{s \approx t}{s\theta \approx t\theta} \qquad \frac{s \approx t}{f(\dots s \dots) \approx f(\dots t \dots)}$$

In other words, we are allowed to substitute arbitrary terms for variables on both sides of an equation $s \approx t$, thereby generating an instance $s\theta \approx t\theta$ of the equation. Here θ is a substitution, a map $\text{Var} \rightarrow \mathcal{T}$.

Moreover, if we have $s \approx t$ then we can replace s by t in an arbitrary term and obtain another equation (the ellipses here are sloppy notation for the same sequence of terms on both sides).

Exercise 12. *Check which of these rules were used in the equational proofs given so far.*

Soundness

Definition 6. *e is a syntactic consequence of E if e can be derived from E by finitely many applications of these rules.*

Notation:

$$E \vdash e$$

Theorem 1. *The deduction system for equations is sound: every syntactic consequence of E is also a semantic consequence.*

Proof. Straightforward induction on the length of the deduction. □

Exercise 13. *Carry out the details in the soundness proof.*

Completeness

Soundness is just a sanity check: our deduction system contains no rules that would produce invalid conclusions.

But how do we know that no rules are missing? Suppose $E \models e$. Is there always an equational proof of e from E ? The answer is given by a famous theorem:

Theorem 2. *Birkhoff*

The deduction system for equations is complete: every semantic consequence of E is also a syntactic consequence.

Hence

$$E \models e \iff E \vdash e$$

Computational Consequences

So suppose we have a set of equations E over some graded alphabet Σ .

How do we check whether an equation e is a semantic consequence of E ?

By Birkhoff's theorem, we can simply enumerate all possible equational proofs with axioms in E . If e follows from E we will discover a proof in finitely many steps. Hence, the problem is semi-decidable.

Of course, if were to implement proof search we would not simply generate potential proof sequences blindly but would try to "get from E to e " in some systematic way.

Alas . . .

One seemingly reasonable approach would be try to generate longer and longer terms that ultimately lead to the desired target equation e .

Unfortunately, in the presence of the sufficiently ill-behaved axioms no such simple strategy is going to work: the proof may involve terms of arbitrary size.

Theorem 3. *In general, it is undecidable whether e is a semantic consequence of E .*

Furthermore, undecidability already rears its ugly head when the signature is very simple.

Undecidable Semigroup

Here is one particularly small example, due to G. S. Tsentin.

There are 5 constants, a, b, c, d, e . We omit the associativity axiom and focus on axioms involving the constants.

4 axioms deal with commutation:

$$ac = ca, ad = da, bc = cb, bd = db.$$

The others are a bit strange:

$$abac = abacc, eca = ae, edb = be.$$

It is undecidable whether an equation between ground terms (no free variables) follows from these axioms.

A Little Lemma

So we have a deduction system that is perfect in principle but has computational problems. Indeed, in practice it turns out that using Birkhoff's rules directly often leads to arguments that are quite long and are often very, very tedious to discover – humans are not particularly good at this kind of task.

It helps to have a few other tools lying around that can help in constructing proofs. For example, here is a lemma that helps in establishing equalities in Boolean algebras.

Lemma 2. *Assume (BA). Then the complement \bar{x} is unique in the sense that $x + y = 1$ and $x \cdot y = 0$ implies $y = \bar{x}$.*

So instead of trying to prove $y = \bar{x}$ directly we can attempt to prove $x + y = 1$ and $x \cdot y = 0$ – which may well be easier.

Justification

Of course, we have to prove first that the lemma holds, using only equational reasoning.

Proof. Assume $x + y = 1$ and $x \cdot y = 0$.

$$\begin{aligned}y &= y \cdot 1 \\&= y \cdot (x + \bar{x}) \\&= y \cdot x + y \cdot \bar{x} \\&= x \cdot y + y \cdot \bar{x} \\&= 0 + y \cdot \bar{x} \\&= x \cdot \bar{x} + y \cdot \bar{x} \\&= (x + y) \cdot \bar{x} \\&= 1 \cdot \bar{x} \\&= \bar{x}\end{aligned}$$

Double Negation

Claim 5. $(BA) \models \overline{\overline{x}} = x.$

Proof. By the lemma, we only have to show that x behaves like the complement of \overline{x} .
But

$$\overline{x} + x = x + \overline{x} = 1 \quad \text{and} \quad \overline{x} \cdot x = x \cdot \overline{x} = 0.$$

Done!

□

And so on and so forth.

Exercise 14. *Establish all the other equations for Boolean algebras listed above.*

Huntington

In the 1930's E. V. Huntington discovered a very simple axiomatization for Boolean algebras. The signature is reduced to $+$ and $\bar{}$ (since product can be defined in terms of these two via Morgan's law).

$$x + y = y + x$$

$$(x + y) + z = x + (y + z)$$

$$x = \overline{\overline{x} + y} + \overline{\overline{x} + \overline{y}}$$

According to Huntington's axioms operation plus is associative and commutative, and the relation between plus and complement is regulated by the third axiom.

It is easy to see that the third axiom holds in any Boolean algebra, but it requires more work to establish the converse.

Robbins' Conjecture

In 1933 H. Robbins conjectured that the third axiom can be replaced by the slightly more cryptic:

$$x = \overline{\overline{x + y} + \overline{x + \overline{y}}}$$

Again it is easy to check that this equation holds in any Boolean algebra, but it is far from clear that the opposite direction also works.

It was shown in the 1970's that to prove this conjecture it suffices to show that Robbins' axioms imply the double negation property $\overline{\overline{x}} = x$ but no one knew how to do this.

The EQP Prover

Robbins' Conjecture was not proven until 1996, and then by the automatic prover EQP, not a human. See

<http://www.mcs.anl.gov/home/mccune/ar/robbins>

One annoying feature of this machine generated proof is that it provides no insight: it consists of just 16 steps of equational reasoning, applied to uncomfortably large expressions. The proof search took 8 days on a workstation.

It is a modest challenge even to just pretty-print the proof.

So, don't worry if you have problems finding some of these proofs.

Representing Boolean Algebras

A good source of Boolean algebras is the powerset of a set A . In fact, we can choose any $B \subseteq \text{pow}(A)$ closed under the operations union, intersection and complement and obtain a Boolean algebra. These algebras are called *fields of sets*.

As it turns out, in essence there are no other Boolean algebras.

Theorem 4. *Stone Representation Theorem*

Up to isomorphism, every Boolean algebra is a field of sets.

The proof is quite involved and requires the notion of an ultrafilter, so we'll skip.

Exercise 15. *Prove the representation theorem in the case where the Boolean algebra is finite.*

All versus Two

The two-element Boolean algebra \mathbb{B} is just one particular model of the axioms.

One might wonder what the relationship is between equations that hold in \mathbb{B} versus equations that are universally valid in all Boolean algebras.

Theorem 5. *An equation is valid in \mathbb{B} if, and only if, it is valid in all Boolean algebras.*

Proof. Suppose $s \approx t$ is an equation that fails to hold in some algebra B . By Stone's theorem we may safely assume that $B \subseteq \text{pow}(A)$ is a field of sets.

So we have for some valuation σ that $\llbracket s \rrbracket_\sigma \neq \llbracket t \rrbracket_\sigma \subseteq A$. Without loss of generality assume $a \in \llbracket s \rrbracket_\sigma - \llbracket t \rrbracket_\sigma$.

Define a homomorphism $f : B \rightarrow \mathbf{2}$ by $f(X) = \begin{cases} 1 & \text{if } a \in X, \\ 0 & \text{otherwise.} \end{cases}$

f shows that $s \approx t$ is not valid in \mathbb{B} . □

Term Algebras

There is a natural way to explain equational theories in terms of algebra rather than logic.

Fix some signature Σ and recall that $\mathcal{T} = \mathcal{T}(\text{Var}, \Sigma)$ is the collection of all terms over Var and Σ .

Definition 7. \mathcal{T} is naturally a Σ -algebra via

$$f^{\mathcal{T}}(t_1, \dots, t_k) = f(t_1, \dots, t_k)$$

This is not a typo: we interpret the function symbol f by the operation: “construct a new term with head f ”.

Note that this algebra has no interesting properties whatsoever, even if we are interested in models of, say, the axiom of commutativity it is NOT the case that $s * t = t * s$ in \mathcal{T} .

Congruences

Now suppose \mathcal{A} is some other Σ -algebra. We can define a homomorphism from \mathcal{T} to \mathcal{A} by fixing values for the variables $\text{Var} \subseteq \mathcal{T}$. So there are lots of homomorphisms $h : \mathcal{T} \rightarrow \mathcal{A}$.

The kernel relation \equiv_h of any homomorphism $h : \mathcal{T} \rightarrow \mathcal{A}$ is an \mathcal{A} -congruence.

Now suppose \mathcal{C} is a class of Σ -algebras. Define $\equiv_{\mathcal{C}}$ to be the intersection of all the congruences \equiv_h where $\mathcal{A} \in \mathcal{C}$.

Then $\equiv_{\mathcal{C}}$ is itself a congruence and we can consider the the quotient structure $\mathcal{T} / \equiv_{\mathcal{C}}$. If we think of $\equiv_{\mathcal{C}}$ as a set of identities (rather than an equivalence relation on \mathcal{T}) it turns out to be none other than the theory of \mathcal{C} .

A natural question is to ask under what conditions this quotient structure belongs itself to \mathcal{C} .

Algebraic Closure

Lemma 3. *Assume that class \mathcal{C} is closed under isomorphisms, subalgebras and products. Then $\mathcal{T} / \equiv_{\mathcal{C}}$ is a free \mathcal{C} -algebra.*

If we slightly strengthen the conditions we obtain a very important type of algebraic class.

Definition 8. A **variety** or **HSP-class** is a class of algebras that is closed under homomorphisms, subalgebras and products.

There is a famous theorem by Birkhoff that connects HSP-classes with equational theorems.

Theorem 6. *Birkhoff*

A class is a variety if, and only if, it is the class of models of an equational theory.

Rewrite Systems

Directed Equations

From the point of view of actual computation, there is one central objection to Birkhoff's system: we don't know which direction an identity $s \approx t$ should be applied in. It may well happen that during the proof of a short equation we need produce intermediate results that are quite large.

So how do we tackle derivability efficiently, at least in some cases?

One way to obtain good algorithms in some cases is to replace the equations by *directed equations*:

$$s \approx t \text{ is replaced by } s \rightarrow t$$

Substituting s by t is always sound, but we may lose completeness: arguments that require the reverse substitution $t \rightarrow s$ are now impossible.

Rewrite Systems

The hope is that we can come up with a collection of *rewrite rules* that make it unnecessary to worry about the proper order and direction of application.

Instead of trying to derive an equation

$$s \approx t$$

we rewrite both s and t into a “normal form” s' and t' .

In the end we simply check if $s' = t'$. The latter is equality of terms and is trivial to check.

Of course, there are several problems with this idea.

Example: Monoids

Suppose we wish to express algebraic simplification in a monoid as a set of rewrite rules. The appropriate language is $\mathcal{L}(*, 1)$ with signature $(2, 0)$.

Among others, we adopt the rule (really just a pair of terms)

$$x * 1 \rightarrow x$$

to express the fact that 1 is a neutral element (on the right).

In order to apply this rule we need to operate on subterms as in

$$y * (x * 1) \rightarrow y * x$$

and we need to be able to substitute terms for the variable x :

$$(y * z) * 1 \rightarrow y * z$$

Contexts and Substitutions

Fix some graded alphabet Σ and a set Var of variables; let \mathcal{T} be the corresponding term algebra.

To make the idea of rewrite rules more precise, write $u[s]_p$ for the term obtained from u by replacing the subterm in position p by s (replacement in context). Other occurrences of s in u are not affected by this operation.

If we think of the term u as a parse tree, the position of a subterm can be specified as a sequence of natural numbers: the sequence expresses a path from the root to some node t . The term (tree) s then replaces the subtree rooted at t .

Exercise 16. *Find a way to represent terms over, say, the graded alphabet for groups and implement this process of substitution.*

Rewrite Relations

We can now explain axiomatically what is meant by a general rewrite relation.

Definition 9.

A binary relation ρ on \mathcal{T} is **compatible** if $s \rho t$ implies that $u[s]_p \rho u[t]_p$.

The relation is **stable** if $s \rho t$ implies that $s\theta \rho t\theta$ for any substitution θ .

ρ is a **rewrite relation** if it is compatible and stable.

We are mostly interested in rewrite relations that have a nice finite description in terms of a few basic rules: construct the least rewrite relation that encompasses the pairs on the given list.

Finite Rewrite Systems

Suppose we have some finite binary relation on \mathcal{T} , i.e. a list of pairs of terms:

$$\mathcal{R} = \{(s_1, t_1), (s_2, t_2), \dots, (s_n, t_n)\}$$

We can define an associated rewrite relation as follows. First, a one-step relation (often called one-step reduction):

$$u[s\theta]_p \rightarrow_{\mathcal{R}} u[t\theta]_p$$

whenever $(s, t) \in \mathcal{R}$ and θ is some substitution. So we pick a subterm in u that is an instance of s and replace it by the corresponding instance of t .

A single step is usually not interesting, we need to iterate. Let $\xrightarrow{+}_{\mathcal{R}}$ be its transitive closure of $\rightarrow_{\mathcal{R}}$ and $\xrightarrow{*}_{\mathcal{R}}$ is the transitive reflexive closure.

We omit the subscript \mathcal{R} whenever possible.

Example: Monoids

As an example, consider the following r/w-system \mathcal{M} for monoids. The language is $\mathcal{L}(*, 1)$ with signature $(2, 0)$. \mathcal{M} has three rules:

$$(x * y) * z \rightarrow x * (y * z)$$

$$x * 1 \rightarrow x$$

$$1 * x \rightarrow x$$

Then we have the following reduction:

$$((1 * (a * 1)) * (1 * a)) * b \xrightarrow{+} a * (a * b)$$

Note that no further reduction steps are possible at this point: none of the rules apply to $a * (a * b)$. This is no coincidence, starting at any term in this system we will ultimately be unable to apply any more reduction steps: eliminate all products involving 1 and move the parens to the right.

Normal Forms

Definition 10. *A term s is **irreducible** or in **normal form** if there exists no term t such that $s \rightarrow_{\mathcal{R}} t$.*

As the monoid example suggests, we would like for each term s to have $s \xrightarrow{*} s'$ where s' is irreducible. In fact, it would be nice if there were exactly one such s' .

There are several possible obstructions to this goal.

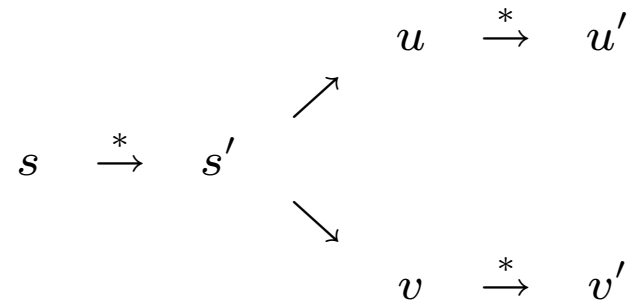
First, it could happen that we just keep going forever

$$s = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n \rightarrow \dots$$

The algorithm trying to find the irreducible form simply fails to terminate.

Forking

Second, since \rightarrow is usually nondeterministic, the reduction may fork into two chains that never again merge.



Even if the chains starting at u and v are both finite we don't have a unique normal form: there is no reason in general why u' should be the same as v' .

Confluence and Termination

In order to rule out these problems we need to impose further conditions on the r/w-system.

Definition 11.

\mathcal{R} is **confluent** if for any $s, t_1, t_2 \in \mathcal{T}$ such that $s \rightarrow t_1$ and $s \rightarrow t_2$ there exists a term t such that $t_1 \xrightarrow{*} t$ and $t_2 \xrightarrow{*} t$.

\mathcal{R} is **terminating** or **noetherian** if for there are no infinite rewrite chains

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_n \rightarrow \dots$$

Note that it is far from clear how one would test whether a given r/w-system has these properties.

Example: Monoids

Claim 6. *The r/w-system \mathcal{M} for monoids is confluent and terminating.*

Proof.

Recall that the rule are

$$(x * y) * z \rightarrow x * (y * z)$$
$$x * 1 \rightarrow x \quad 1 * x \rightarrow x$$

For termination first note that the unit-removal rules $x * 1 \rightarrow x, 1 * x \rightarrow x$ can be applied only as many times as the term contains the constant 1: there is no way another 1 can be introduced.

Now note that the unit-removal rules commute with the associativity rule. Hence we can move all applications of unit-removal rules to the front of the rewrite sequence. So let's assume t contains no occurrence of 1 and we only apply the associativity rule $(x * y) * z \rightarrow x * (y * z)$.

Termination

Here is a slightly overkillish but cute proof for termination. Recall that a full binary tree T can be coded as a binary sequence $\lambda(T)$ as follows (this method pushes right up against the information theoretic bound).

- A single-node tree is coded as 0.
- A tree with left subtree a and right subtree b is coded as $1 \lambda(a) \lambda(b)$

Hence if T has n leaves then $\#_0\lambda(T) = n - 1$ and $\#_1\lambda(T) = n$.

It so happens that if T' is the result of applying the associativity rule to T then $\lambda(T') < \lambda(T)$.

Hence we must have termination.

Confluence

For confluence use induction on the size of the tree to show the following. Let a_1, a_2, \dots, a_n the frontier of the tree of t (from left to right).

Then t will be rewritten to

$$a_1 * (a_2 * (\dots (a_{n-1} * a_n) \dots))$$

□

Exercise 17. *Show that the tree coding used in termination really works as advertised.*

Exercise 18. *Fill in the details for the confluence argument.*

Normal Form Systems

The combination of confluence and termination is all we need for unique normalization.

Lemma 4. *If a rewrite system is confluent and terminating, then every term has a unique normal form.*

Proof. It follows from termination that for some irreducible t we have $s \xrightarrow{*} t$.

So suppose that t_1 and t_2 are irreducible and $s \xrightarrow{*} t_1$, $s \xrightarrow{*} t_2$. By confluence there is some term t such that $t_1 \xrightarrow{*} t$ and $t_2 \xrightarrow{*} t$. By irreducibility, this implies $t_1 = t_2$.

□

So we can blindly apply the rewrite rules and we will always arrive at the same irreducible term.

Of course, this ignores questions of efficiency: the number of steps needed to get to irreducibility may well vary.

Equations versus R/W

Let \leftrightarrow^* be the reflexive, symmetric and transitive closure of \rightarrow . Then $s \leftrightarrow^* t$ means that we can rewrite s to t by using the rules in \mathcal{R} in both directions, as if they were equations rather than directed equations.

So suppose \mathcal{E} is some system of equations over \mathcal{T} . Impose an arbitrary direction on these equations to obtain a rewrite system \mathcal{R} .

Lemma 5. $\mathcal{E} \vdash s \approx t$ if, and only if, $s \leftrightarrow^* t$.

Note that \leftrightarrow^* is an equivalence relation and partitions \mathcal{T} into classes of “rewriteable” terms.

If we have confluence and termination, we have a canonical representative for each of these equivalence classes.

Exercise 19. *Prove the last lemma.*

Exercises

Exercise 20. *Show that a rewrite system that has a rule $x \rightarrow t$ where x is a variable cannot be terminating. Show that a rewrite system that has a rule $s \rightarrow t$ where the variables in t are not a subset of the variables in s cannot be terminating.*

Exercise 21. *Suppose \mathcal{R} is a rewrite system so that for every rule $s \rightarrow t$ we have $|s| > |t|$ and for every variable x : $|s|_x \geq |t|_x$. Show that \mathcal{R} must be terminating.*

Exercise 22. *Add two constants a and b to \mathcal{M} . Argue that the new system represents all words over a two-letter alphabet. What happens if we add one more rule $b * a \rightarrow b * b$?*

Exercise 23. *Show how to test whether a particular rule $s \rightarrow t \in \mathcal{R}$ is applicable to a term u : is there a position p in u (i.e. a subterm of u) and a substitution θ such that*

$$u = u[s\theta]_p.$$

Example: Group Theory

Augment the three rules for monoids by the following (in the extended language $\mathcal{L}(*, ^{-1}, 1)$ of signature $(2, 1, 0)$). Rewrite system \mathcal{G} :

$$1^{-1} \rightarrow 1$$

$$x^{-1} * x \rightarrow 1$$

$$x * x^{-1} \rightarrow 1$$

$$(x^{-1})^{-1} \rightarrow 1$$

$$x * (x^{-1} * y) \rightarrow y$$

$$x^{-1} * (x * y) \rightarrow y$$

$$(x * y)^{-1} \rightarrow y^{-1} * x^{-1}$$

Exercise 24. *Determine whether \mathcal{G} is terminating and/or confluent.*

Church-Rosser

Confluence means: any forking reduction can be made to merge again later.

There is a property closely related to confluence that is important in the theory of r/w-systems.

Definition 12. *A rewrite system is Church-Rosser if for any $s, t \in \mathcal{T}$ such that $s \xleftrightarrow{*} t$ there exists a term r such that $s \xrightarrow{*} r$ and $t \xrightarrow{*} r$.*

Lemma 6. *A rewrite system is confluent if, and only if, it has the Church-Rosser property.*

Exercise 25. *Prove the lemma (this is an exercise in induction).*

Decision Problems

There are several important decision problems associated with any finite rewrite system \mathcal{R} .

Problem: **Common Ancestor Problem**

Instance: Two terms s and t .

Question: Is there a common ancestor $r \xrightarrow{*} s$ and $r \xrightarrow{*} t$?

Problem: **Common Descendant Problem**

Instance: Two terms s and t .

Question: Is there a common descendant $s \xrightarrow{*} r$ and $t \xrightarrow{*} r$?

Problem: **Word Problem**

Instance: Two terms s and t .

Question: Are s and t equivalent under $\xleftrightarrow{*}$?

The terminology for the third problem goes back to the famous word problem for finitely presented groups.

Uniform Versions

Note that we have fixed the rewrite system for these problems. Technically, this is known as the *non-uniform version* of the problem.

There is an obvious generalization, the *uniform version* of the problem, where \mathcal{R} is part of the input.

So, we may be interested in the Word Problem for a specific rewrite system \mathcal{R} (which may be designed to describe, say, a specific group) or we may try to tackle this problem uniformly for all possible rewrite systems or at least for all rewrite systems in a certain class.

Needless to say, the uniform version is harder.

In fact, even if the non-uniform version is decidable for each rewrite system in some class, the uniform version may well not be: there may be no effective way to derive the appropriate decision algorithm from the system.

Testing Termination

How can we check if \mathcal{R} is terminating? In general, we cannot.

Theorem 7. *It is undecidable whether a finite rewrite system is terminating.*

Proof.

Encode the workings of a Turing machine as a rewrite system.

□

Note that termination is undecidable even for a single fixed starting term (corresponding to, say, the computation of a Turing machine on empty tape).

Ordering Terms

But in concrete cases, motivated by an attempt to model some specific mathematical structure, one can often get by associating a measure of complexity to each term s . The natural expectation is that each rewrite step reduces the complexity of the term.

Definition 13. A **reduction ordering** is a strict pre-order \triangleleft on \mathcal{T} that is stable and compatible.

Of course, we are interested in reduction orderings that extend the one-step relation (actually, its converse) of the r/w-system in question:

$$s \rightarrow t \quad \text{implies} \quad s \triangleright t$$

If we can find such a reduction order that is also well-founded then there cannot be infinite chains of reductions.

Note that ordinary length of terms is compatible and well-founded but not stable.

Reduction Orders and Termination

Lemma 7. *A rewrite system \mathcal{R} is terminating if, and only if, it has a well-founded reduction order.*

Proof.

If \mathcal{R} is terminating we can use the converse of $\xrightarrow{+}$ as the reduction order.

In the opposite direction, consider a rewrite step $s_1 \rightarrow s_2$. Then $s_1 = t[u\theta]_p$ and $s_2 = t[v\theta]$ where $(u, v) \in \mathcal{R}$ for some substitution θ and some term t . Since the order is compatible and stable, reduction chains in \mathcal{R} translate into strictly descending chains in the order. Since the order is well-founded we are done.

□

Exercise 26. *Construct a well-founded reduction order for the r/w-system \mathcal{M} .*

Unification

Unification

For the critical pair method below we need to find a substitution that identifies to terms.

Definition 14. *Given two terms s and t with free variables among x_1, \dots, x_n , a **unifier** for s and t is a substitution $\theta : \{x_1, \dots, x_n\} \rightarrow \mathcal{T}$ such that $s\theta = t\theta$.*

For example, in the language of rings let

$$s = (x + y) \cdot x \qquad t = (y + x) \cdot y$$

There are many unifiers, e.g. $(1/x, 1/y)$, $(x + 1/x, x + 1/y)$ and so on.

But one of these is the most general: $\theta = (x/x, x/y)$. Most general means that every unifier σ is already an instance of θ : $\sigma = \tau \circ \theta$.

This is the **most general unifier (MGU)** for s and t .

Robinson 1965

It was shown by A. Robinson that

- If two terms are unifiable at all then they also have a MGU.
- In which case the most general unifier is unique.
- There is an algorithm to test unifiability. The algorithm returns the MGU if it exists, No otherwise.

The idea is that one scans s and t from left to right (assuming prefix form) and finds the first subterms s' and t' where s and t disagree. These are called critical subterms.

In order to be able to unify s' and t' the critical subterm condition (CSC) must be satisfied:

- one, say s' , is a variable x , and
- the other is some term not containing x .

In which case $x \mapsto t'$ is part of the substitution; otherwise unification has failed.

Iterate the Critical Subterm Method

The whole algorithm consists of repeated applications of this basic step.

```
theta = identity;
while( there are critical subterms s', t' )
  if( CSC is satisfied )
    then
      apply substitution x/t' to s and t
      apply substitution x/t' to theta
    else
      return NO;
return theta;
```

Exercise 27. *Implement this algorithm (a high level language such as Lisp or Mathematica is highly recommended).*

Application: Logic Programming

Logic programming languages such as PROLOG rely heavily on Horn clauses since the more general case is prohibitively complex.

Note that to get real mileage out of this machinery one needs unification.

The problem is that one has to deal with more than just propositional variables: there will be terms such as

$$R(x, f(y)) \rightarrow S(x)$$

that express some relationship and/or operations on a domain of individuals (R is a binary relation symbol, S a unary relation symbol and f a unary function).

The problem now is that the “literals” involve patterns and one has to work harder to make things match up.

Here is just one example and a little application.

Unification

Suppose we have clauses

$$T(w) \rightarrow R(g(z), w), R(x, f(y)) \rightarrow S(x)$$

We would like to conclude that from T one can infer S , but one must take the relation and function symbols into account, and the variables.

The way to handle this is by finding a unifier, a substitution θ , such that

$$R(g(z), w)[\theta] = R(x, f(y))[\theta]$$

In this case $x \mapsto g(z), w \mapsto f(y)$ works.

Applying the Unifier

Substitution $x \mapsto g(z), w \mapsto f(y)$ produces

$$R(x, f(y))[\theta] = R(g(z), f(y)) = R(g(z), w)[\theta]$$

Once the unifier is applied the literals match up directly and we get

$$T(f(y)) \rightarrow R(g(z), f(y)), R(g(z), f(y)) \rightarrow S(g(z))$$

from which we can conclude

$$T(f(y)) \rightarrow S(g(z))$$

Formalizing Arithmetic

Using such general terms allows one to perform computations as proofs.

$$\begin{aligned} & \rightarrow \text{add}(x, 0, x) \\ \text{add}(x, y, z) & \rightarrow \text{add}(x, y^+, z^+) \\ & \rightarrow \text{mult}(x, 0, 0) \\ \text{mult}(x, y, t), \text{add}(t, x, z) & \rightarrow \text{mult}(x, y^+, z) \end{aligned}$$

Note that these are really just the primitive recursive definitions of addition and multiplication.

Verifying Arithmetic

From these axioms Φ we can prove assertions such as

$$\text{mult}(0^{++}, 0^{+++}, 0^{++++++})$$

Note that since we can only deal with refutations here, the right way to do this is to show that

$$\Phi \cup \left\{ \text{mult}(0^{++}, 0^{+++}, 0^{++++++}) \rightarrow \perp \right\}$$

is a contradiction.

This particular result is admittedly not too impressive, but the method shows a deep connection between computation and logic.

This connection was first made very explicit in Hilbert's 23 Problems in 1900 at the Mathematician's Congress in Paris.

Getting Values

Writing $\text{res}(z)$ for “result” we can even ask the system to show that $\text{res}(z)$ holds for some z , given our axioms: it has to refute

$$\Phi \cup \left\{ \text{mult}(0^{++}, 0^{+++}, z) \rightarrow \text{res}(z), \text{res}(z) \rightarrow \perp \right\}$$

In fact, by keeping track of the unifiers we can get a concrete value for z , in this case $6 = 0^{+++++}$.

So the refutation produces the correct value of the computation.

Knuth-Bendix

Suppose we have a terminating r/w-system \mathcal{R} . How do we check confluence?

Note that the problem is not even obviously semi-decidable: it seems to involve a search over all forking reductions followed by an application of the Common Descendant Problem.

If two rules have non-overlapping handles (which we won't define precisely here) then they coexist: we can apply them in arbitrary order without any problem.

Potential obstructions to confluence come from the pairs of rules that overlap in a sense. We need to make this precise.

Consider two rules

$$s_1 \rightarrow t_1 \quad \text{and} \quad s_2 \rightarrow t_2$$

where, without loss of generality, the variables in s_1 are distinct from the variable in s_2 .

Critical Pairs

Now suppose s_1 has a subterm s' , not a variable, such that s' and s_2 are unifiable.

Let θ be a MGU.

Then $s_1\theta$ admits two different applications of the rules:

- We can replace all of $s_1\theta$ by $t_1\theta$ according to rule 1.
- We can replace only $s'\theta$ by $t_2\theta$ according to rule 2.

The resulting two terms are a *critical pair*.

Theorem 8. *Let \mathcal{R} be terminating. The \mathcal{R} is confluent if it is confluent on all critical pairs.*

Summary

- Propositional logic is too weak for many applications; equational logic is a first step towards a more powerful system.
- Birkhoff's system provides a formalization of equational logic.
- For efficiency reasons, rewrite systems based on directed equations are the tool of choice in applications.
- Unification is a standard tool used in many places in theorem proving and logic programming.