MACHINE REASONING

1. Resolution Theorem Proving

Perhaps the most popular deduction rule in automated theorem proving is

$$\begin{array}{c} \text{A or B} \\ \neg\,\text{A or C} \\ \hline \therefore\ \text{B or C} \end{array}$$

This rule, promoted by J.A. Robinson in the mid 1960's, is called <u>resolution</u>.

Let's look at an example of how a machine might use this rule.

Suppose the machine has the following facts:

(1) $x \cdot x + y \cdot y > 0$ implies $x > 0$ or $y > 0$

(2) $u + u \cdot v > 0$ or $u \approx 0$.

First we put this information into the form of <u>clauses</u>. (A clause is a disjunction of <u>literals</u>.)

(1) $\neg(x \cdot x + y \cdot y > 0)$ or $x > 0$ or $y > 0$

(2) $u + u \cdot v > 0$ or $u \approx 0$.

One cannot immediately apply resolution as there is no matching pair A and $\neg$A to cancel.

However special case of (1) and (2), obtained by substitution, might be in the correct form to apply the rule.

We will have to use the first literal of (1) to have a negation symbol.

Observe that no substitution will turn

$$\neg(x \cdot x + y \cdot y > 0) \quad \text{and} \quad u \approx 0$$

into a <u>complementary</u> pair of literals.

However the substitution

$$\begin{array}{l|l} x \leftarrow w & u \leftarrow w \cdot w \\ y \leftarrow w \cdot w & v \leftarrow w \cdot w \end{array}$$

turns	into
$\neg\,(x \cdot x + y \cdot y > 0)$	$\neg\,[w \cdot w + (w \cdot w) \cdot (w \cdot w) > 0]$
$u + v \cdot w > 0$	$w \cdot w + (w \cdot w) \cdot (w \cdot w) > 0$,

a complementary pair of literals. This substitution gives the following special cases of (1) and (2), where a repeat of a literal $w \cdot w > 0$ has been deleted:

$$\neg\,[w \cdot w + (w \cdot w) \cdot (w \cdot w) > 0] \quad \text{or } w \cdot w > 0$$

$$w \cdot w + (w \cdot w) \cdot (w \cdot w) > 0 \quad \text{or} \quad w \cdot w \approx 0.$$

Now applying resolution to these two clauses gives the result

$$w \cdot w > 0 \quad \text{or} \quad w \cdot w \approx 0.$$

The key question here is: Why not use a substitution like

$$\begin{array}{l|l} x \leftarrow w + z & u \leftarrow (w+z)\cdot(w+z) \\ y \leftarrow (w+z)\cdot(w+z) & v \leftarrow (w+z)\cdot(w+z) \end{array}$$

instead of

$$\begin{array}{l|l} x \leftarrow w & u \leftarrow w \cdot w \\ y \leftarrow w \cdot w & v \leftarrow w \cdot w \end{array} ?$$

The answer is simply that one wants the <u>most general</u> substitution possible, to get the most general conclusion possible.

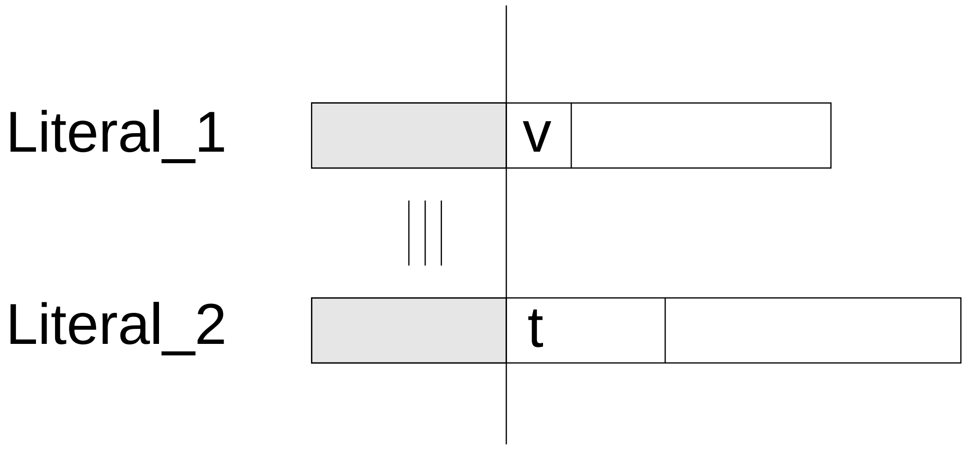
Note that the first substitution is a special case of the second one.

Robinson showed in 1965 that if one has literals $\neg L_1$ and L_2 such that some substitution makes L_1 and L_2 equal, then there is a unique most general substitution that makes them equal.

Or, in the parlance of Computer Science, if L_1 and L_2 have a unifier then they have a unique most general unifier.

Robinson gave an algorithm to determine if two literals could be unified, and if so, the algorithm found the most general unifier.

Let's see how it works on the above example. We assume the literals are expressed in prefix form.



v is a variable t is a term

If v does not occur in t then apply: $v \longleftarrow t$

Let us apply the algorithm to the literals

$$x \bullet x + y \bullet y > 0 \quad \text{and} \quad u + u \bullet v > 0.$$

The initial setup is:

$>$	$+$	$\bullet$	x	x	$\bullet$	y	y	0
$>$	$+$	u	$\bullet$	u	v	0		

Now apply the substitution $u \leftarrow \bullet xx$.

$>$	$+$	$\bullet$	x	x	$\bullet$	y	y	0		
$>$	$+$	$\bullet$	x	x	$\bullet$	$\bullet$	x	x	v	0

Next apply $y \leftarrow \bullet xx$.

$>$	$+$	$\bullet$	x	x	$\bullet$	$\bullet$	x	x	$\bullet$	x	x	0
$>$	$+$	$\bullet$	x	x	$\bullet$	$\bullet$	x	x	v	0		

Finally apply $v \leftarrow \bullet xx$.

$>$	$+$	$\bullet$	x	x	$\bullet$	$\bullet$	x	x	$\bullet$	x	x	0
$>$	$+$	$\bullet$	x	x	$\bullet$	$\bullet$	x	x	$\bullet$	x	x	0

The literals are unifiable, and we have the most general unifier $x \leftarrow x$ and $y, u, v \leftarrow x \bullet x$.

---

2. Normal Form Term Rewrite Systems.

Shortly after Robinson's work, and independent of it, Knuth discovered the same algorithm in his work on term rewrite systems.

Let us consider the following equations defining groups:

$$\begin{aligned} ex &\approx x \\ x^{-1}x &\approx e \\ (xy)z &\approx x(yz). \end{aligned}$$

Knuth and Bendix (1967) noted that the following is a normal form TRS for groups:

1.	$e^{-1} \rightarrow e$	6.	$xx^{-1} \rightarrow e$
2.	$xe \rightarrow x$	7.	$x^{-1}(xy) \rightarrow y$
3.	$ex \rightarrow x$	8.	$x(x^{-1}y) \rightarrow y$
4.	$(x^{-1})^{-1} \rightarrow x$	9.	$(xy)^{-1} \rightarrow y^{-1}x^{-1}$
5.	$x^{-1}x \rightarrow e$	10.	$(xy)z \rightarrow x(yz)$.

If one takes a term in the language of groups, say $x((y^{-1}z)x)^{-1}$, and repeatedly applies these rules, in any order that you like, you will eventually end up with the unique normal form $z^{-1}y$.

The problem that Knuth and Bendix tackled was how to get from the set of 3 equations to the set of 10 rewrite rules.

Their first step was to wisely choose a well-founded generic ordering $>$ on the terms that was preserved under substitution and replacement. Their generic choices were based on weighted lexicographical orderings of the terms.

Using this they <u>oriented</u> the defining equations as follows:

$$\begin{array}{lrcl} (1) & ex & \to & x \\ (2) & y^{-1}y & \to & e \\ (3) & (uv)w & \to & u(vw). \end{array}$$

You don't have to look far to see that this does not give normal forms. For example xe does not reduce in this system to its normal form x.

So Knuth and Bendix wanted to find good candidates for more rewrite rules. Their idea was to use unification to look for divergence when applying the rules obtained so far.

For example, consider rules (2) and (3) and find a most general unifier of $y^{-1}y$ with uv. The answer is simply

$$\begin{array}{rcl} y & \leftarrow & z \\ u & \leftarrow & z^{-1} \\ v & \leftarrow & z. \end{array}$$

Under this substitution rules (2) and (3) become

$$\begin{array}{lrcl} (2') & z^{-1}z & \rightarrow & e \\ (3') & (z^{-1}z)w & \rightarrow & z^{-1}(zw). \end{array}$$

Applying (2′) to the left side of (3′), and then reducing further where possible, gives the following result:

$$\begin{tikzcd}
& (z^{-1}z)w \arrow[dl] \arrow[dr] & & \\
z^{-1}(zw) & & ew \arrow[dr] & \\
& & & w
\end{tikzcd}$$

Thus we see that $(z^{-1}z)w$ reduces to two different terms, namely $z^{-1}(zw)$ and w. One of them is clearly simpler than the other. This suggests that we add the new reduction rule:

$$(4) \quad z^{-1}(zw) \;\rightarrow\; w.$$

By repeating this procedure, with appropriate tidying of the collection of working rules along the way, one arrives at the 10 rules for groups.

The use of most general unifiers leads to a halt in the Knuth-Bendix procedure if one actually has a normal form TRS for the set of equations.

One can also use the KB procedure to prove theorems in equational logic. Suppose you want to prove $E \vdash s \approx t$. Then one applies the KB procedure to search for a normal form TRS for E.

If one finds such a TRS, then use it to reduce s and t to their normal forms, and see if they are equal.

However, while searching for a normal form TRS one keeps reducing s and t with the reductions obtained so far. If one should by chance reduce them to the same term, then the equation $s \approx t$ is true, and one can abandon the search for a normal form TRS.

This method was used by Robyn Edelson (1990) to find a machine proof that strongly regular rings ($\forall x \exists y\ x^2 y \approx x$) are von Neumann regular ($\forall x \exists y\ xyx \approx x$) , using HIPER, written by Jim Christian (U Texas).

1. $x^2 x^q = x$
2. $x x^q x^{qq} = x$
3. $x(y - x x^q y) = 0$
4. $[(y - x x^q y)x]^2 [(y - x x^q y)x]^q = 0$
5. $[(y - x x^q y)x]^2 [(y - x x^q y)x]^q = (y - x x^q y)x$
6. $(y - x x^q y)x = 0$
7. $(x^{qq} - x)x = 0$
8. $(y + (-y)^{qq}) yx = 0$
9. $(-x)^{qq} x x^q + x = 0$
10. $x^{qq}(-x)(-x)^q = x$
11. $x(-x)(-x)^q = x x^q x$
12. $x = x x^q x$