

# **Automated Reasoning II**

**Uwe Waldmann**

**Summer Term 2024**

# Topics of the Course

---

Decision procedures:

- equality (congruence closure),
- algebraic theories,
- combinations.

Satisfiability modulo theories (SMT):

- CDCL(T),
- dealing with universal quantification.

Superposition:

- combining ordered resolution and completion,
- optimizations,
- integrating theories.

# Part 1: Decision Procedures

---

In general, validity (or unsatisfiability) of first-order formulas is undecidable.

To get decidability results, we have to impose restrictions on

- signatures,
- formulas,
- and/or algebras.

## 1.1 Theories and Fragments

---

So far, we have considered the validity or satisfiability of “unstructured” sets of formulas.

We will now split these sets of formulas into two parts:  
a theory (which we keep fixed) and a set of formulas that we consider relative to the theory.

# Theories and Fragments

---

A **first-order theory**  $\mathcal{T}$  is defined by

its signature  $\Sigma = (\Omega, \Pi)$

its axioms, that is, a set of closed  $\Sigma$ -formulas.

(We often use the same symbol  $\mathcal{T}$  for a theory and its set of axioms.)

Note: This is the *syntactic view* of theories. There is also a *semantic view*, where one specifies a class of  $\Sigma$ -algebras  $\mathcal{M}$  and considers  $Th(\mathcal{M})$ , that is, all closed  $\Sigma$ -formulas that hold in the algebras of  $\mathcal{M}$ .

# Theories and Fragments

---

A  $\Sigma$ -algebra that satisfies all axioms of  $\mathcal{T}$  is called a  $\mathcal{T}$ -algebra (or  $\mathcal{T}$ -interpretation).

$\mathcal{T}$  is called **consistent** if there is at least one  $\mathcal{T}$ -algebra.  
(We will only consider consistent theories.)

# Theories and Fragments

---

We can define models, validity, satisfiability, entailment, equivalence, etc., relative to a theory  $\mathcal{T}$ :

A  $\mathcal{T}$ -algebra that is a model of a  $\Sigma$ -formula  $F$  is also called a  $\mathcal{T}$ -model of  $F$ .

A  $\Sigma$ -formula  $F$  is called  $\mathcal{T}$ -valid,  
if  $\mathcal{A}, \beta \models F$  for all  $\mathcal{T}$ -algebras  $\mathcal{A}$  and assignments  $\beta$ .

A  $\Sigma$ -formula  $F$  is called  $\mathcal{T}$ -satisfiable,  
if  $\mathcal{A}, \beta \models F$  for some  $\mathcal{T}$ -algebra and assignment  $\beta$   
(and otherwise  $\mathcal{T}$ -unsatisfiable).

( $\mathcal{T}$ -satisfiability of sets of formulas,  $\mathcal{T}$ -entailment,  $\mathcal{T}$ -equivalence:  
analogously.)

# Theories and Fragments

---

A *fragment* is some syntactically restricted class of  $\Sigma$ -formulas.

Typical restriction: only certain quantifier prefixes are permitted.

## 1.2 Equality

---

Theory of equality:

Signature: arbitrary

Axioms: none

(but the equality predicate  $\approx$  has a fixed interpretation)

Alternatively:

Signature contains a binary predicate symbol  $\sim$  instead of the built-in  $\approx$

Axioms: reflexivity, symmetry, transitivity, congruence for  $\sim$

# Equality

---

In general, satisfiability of first-order formulas w. r. t. equality is undecidable.

However, we will show that it is decidable for *ground* first-order formulas.

Note: It suffices to consider conjunctions of literals.

Arbitrary ground formulas can be converted into DNF;

a formula in DNF is satisfiable if and only if one of its conjunctions is satisfiable.

# Equality

---

Note that our problem can be written in several ways:

An equational clause

$\forall \vec{x} (A_1 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee \neg B_k)$  is  $\mathcal{T}$ -valid

iff

$\exists \vec{x} (\neg A_1 \wedge \dots \wedge \neg A_n \wedge B_1 \wedge \dots \wedge B_k)$  is  $\mathcal{T}$ -unsatisfiable

iff

the Skolemized (ground!) formula

$(\neg A_1 \wedge \dots \wedge \neg A_n \wedge B_1 \wedge \dots \wedge B_k)\{\vec{x} \mapsto \vec{c}\}$  is  $\mathcal{T}$ -unsatisfiable

iff

$(A_1 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee \neg B_k)\{\vec{x} \mapsto \vec{c}\}$  is  $\mathcal{T}$ -valid

# Equality

---

Other names:

The theory is also known as **EU**F (equality with uninterpreted function symbols).

The decision procedures for the ground fragment are called **congruence closure** algorithms.

# Congruence Closure

---

Goal: check (un-)satisfiability of a ground conjunction

$$u_1 \approx v_1 \wedge \dots \wedge u_n \approx v_n \wedge \neg s_1 \approx t_1 \wedge \dots \wedge \neg s_k \approx t_k$$

Idea:

transform  $E = \{u_1 \approx v_1, \dots, u_n \approx v_n\}$  into an equivalent convergent TRS  $R$  and check whether  $s_i \downarrow_R = t_i \downarrow_R$ .

if  $s_i \downarrow_R = t_i \downarrow_R$  for some  $i$ :

$$s_i \downarrow_R = t_i \downarrow_R \Leftrightarrow s_i \leftrightarrow_E^* t_i \Leftrightarrow E \models s_i \approx t_i \Rightarrow \text{unsat.}$$

if  $s_i \downarrow_R = t_i \downarrow_R$  for no  $i$ :

$$T_{\Sigma}(X)/R = T_{\Sigma}(X)/E \text{ is a model of the conjunction } \Rightarrow \text{sat.}$$

# Congruence Closure

---

In principle, one could use Knuth-Bendix completion to convert  $E$  into an equivalent convergent TRS  $R$ .

If done properly (see exercises), Knuth-Bendix completion terminates for ground inputs.

However, for the ground case, one can optimize the general procedure.

# Congruence Closure

---

First step:

Flatten terms:

Introduce new constant symbols  $c_1, c_2, \dots$  for all subterms:

$$g(a, h(h(b))) \approx h(a)$$

is replaced by

$$a \approx c_1 \wedge b \approx c_2 \wedge h(c_2) \approx c_3 \wedge h(c_3) \approx c_4$$

$$\wedge g(c_1, c_4) \approx c_5 \wedge h(c_1) \approx c_6 \wedge c_5 \approx c_6$$

# Congruence Closure

---

Result: only two kinds of equations left.

D-equations:  $f(c_{i_1}, \dots, c_{i_n}) \approx c_{i_0}$  for  $f/n \in \Omega$ ,  $n \geq 0$ .

C-equations:  $c_i \approx c_j$ .

$\Rightarrow$  efficient indexing (e. g., using hash tables),  
obvious termination for D-equations.

# Inference Rules

---

The congruence closure algorithm is presented as a set of inference rules working on a set of equations  $E$  and a set of rules  $R$ :

$$E_0, R_0 \vdash E_1, R_1 \vdash E_2, R_2 \vdash \dots$$

At the beginning,  $E = E_0$  is the set of C-equations and  $R = R_0$  is the set of D-equations oriented left-to-right. At the end,  $E$  should be empty; then  $R$  is the result.

Notation: The formula  $s \overset{\cdot}{\approx} t$  denotes either  $s \approx t$  or  $t \approx s$ .

# Inference Rules

---

*Simplify:*

$$\frac{E \cup \{c \dot{\approx} c'\}, R \cup \{c \rightarrow c''\}}{E \cup \{c'' \dot{\approx} c'\}, R \cup \{c \rightarrow c''\}}$$

*Delete:*

$$\frac{E \cup \{c \approx c\}, R}{E, R}$$

*Orient:*

$$\frac{E \cup \{c \dot{\approx} c'\}, R}{E, R \cup \{c \rightarrow c'\}} \quad \text{if } c \succ c'$$

# Inference Rules

---

*Collapse:*

$$\frac{E, R \cup \{t[c]_p \rightarrow c', c \rightarrow c''\}}{E, R \cup \{t[c'']_p \rightarrow c', c \rightarrow c''\}} \quad \text{if } p \neq \varepsilon$$

*Deduce:*

$$\frac{E, R \cup \{t \rightarrow c, t \rightarrow c'\}}{E \cup \{c \approx c'\}, R \cup \{t \rightarrow c\}}$$

Note: for ground rewrite rules, critical pair computation does not involve substitution. Therefore, every critical pair computation can be replaced by a simplification, either using Deduce or Collapse.

# Inference Rules

---

Theorem 1.1:

Let  $E_0$  be a finite set of C-equations, let  $R_0$  be a finite set of D-equations oriented left-to-right w.r.t.  $\succ$ , and let  $\succ$  be a total ordering on constants. Then the inference system terminates with a final state  $(E_n, R_n)$  where  $E_n = \emptyset$ ,  $R_n$  is terminating and confluent, and  $\approx_{E_0 \cup R_0}$  equals  $\approx_{R_n}$ .

# Strategy

---

The inference rules are applied according to the following strategy:

- (1) If there is an equation in  $E$ , use Simplify as long as possible for this equation, then use either Delete or Orient. Repeat until  $E$  is empty.
- (2) If Collapse is applicable, apply it, if now Deduce is applicable, apply it as well. Repeat until Collapse is no longer applicable.
- (3) If  $E$  is non-empty, go to (1), otherwise return  $R$ .

# Implementation

---

Instead of fixing the ordering  $\succ$  in advance, it is preferable to define it on the fly during the algorithm:

If we orient an equation  $c \approx c'$  between two constant symbols, we try to make that constant symbol larger that occurs less often in  $R$   
 $\Rightarrow$  fewer Collapse steps.

Additionally:

Use various index data structures so that all the required operations can be performed efficiently.

Use a union-find data structure to represent the equivalence classes encoded by the C-rules.

# Implementation

---

Average runtime for an implementation using hash tables:

$O(m \log m)$ , where  $m$  is the number of edges in the graph representation of the initial C and D-equations.

## One Small Problem

---

The inference rules are sound in the usual sense: The conclusions are entailed by the premises, so every  $\mathcal{T}$ -model of the premises is a  $\mathcal{T}$ -model of the conclusions.

For the initial flattening, however, we get a weaker result: We have to *extend* the  $\mathcal{T}$ -models of the original equations to obtain models of the flattened equations.

That is, we get a new algebra with the same universe as the old one, with the same interpretations for old functions and predicate symbols, but with appropriately chosen interpretations for the new constants.

## One Small Problem

---

Consequently, the relations  $\approx_E$  and  $\approx_R$  for the original  $E$  and the final  $R$  are not the same. For instance,  $c_3 \approx_E c_7$  does not hold, but  $c_3 \approx_R c_7$  may hold.

On the other hand, the model extension preserves the universe and the interpretations for old symbols. Therefore, if  $s$  and  $t$  are terms over the old symbols, we have  $s \approx_E t$  iff  $s \approx_R t$ .

This is sufficient for our purposes: The terms  $s_i$  and  $t_i$  that we want to normalize using  $R$  do not contain new symbols.

## Other Predicate Symbols

---

If the initial ground conjunction contains also non-equational literals  $[\neg] P(t_1, \dots, t_n)$ , treat these like equational literals  $[\neg] P(t_1, \dots, t_n) \approx true$ .  
Then use the same algorithm as before.

# History

---

Congruence closure algorithms have been published, among others, by Shostak (1978). by Nelson and Oppen (1980), and by Downey, Sethi and Tarjan (1980).

Kapur (1997) showed that Shostak's algorithm can be described as a completion procedure.

Bachmair and Tiwari (2000) did this also for the Nelson/Oppen and the Downey/Sethi/Tarjan algorithm.

The algorithm presented here is the Downey/Sethi/Tarjan algorithm in the presentation of Bachmair and Tiwari.

## 1.3 Linear Rational Arithmetic

---

There are several ways to define **linear rational arithmetic**.

We need at least the following signature:

$$\Sigma = (\{0/0, 1/0, +/2\}, \{</2\})$$

and the pre-defined binary predicate  $\approx$ .

# Linear Rational Arithmetic

---

The equational part of linear rational arithmetic is described by the theory of **divisible torsion-free abelian groups**:

$$\forall x, y, z (x + (y + z) \approx (x + y) + z) \quad (\text{associativity})$$

$$\forall x, y (x + y \approx y + x) \quad (\text{commutativity})$$

$$\forall x (x + 0 \approx x) \quad (\text{identity})$$

$$\forall x \exists y (x + y \approx 0) \quad (\text{inverse})$$

$$\text{For all } n \geq 1: \forall x (\underbrace{x + \dots + x}_{n \text{ times}} \approx 0 \rightarrow x \approx 0) \quad (\text{torsion-freeness})$$

$$\text{For all } n \geq 1: \forall x \exists y (\underbrace{y + \dots + y}_{n \text{ times}} \approx x) \quad (\text{divisibility})$$

$$\neg 1 \approx 0 \quad (\text{non-triviality})$$

# Linear Rational Arithmetic

---

Note: Quantification over natural numbers is not part of our language. We really need infinitely many axioms for torsion-freeness and divisibility.

# Linear Rational Arithmetic

---

By adding the axioms of a compatible strict total ordering, we define **ordered divisible abelian groups**:

$$\forall x (\neg x < x) \quad (\text{irreflexivity})$$

$$\forall x, y, z (x < y \wedge y < z \rightarrow x < z) \quad (\text{transitivity})$$

$$\forall x, y (x < y \vee y < x \vee x \approx y) \quad (\text{totality})$$

$$\forall x, y, z (x < y \rightarrow x + z < y + z) \quad (\text{compatibility})$$

$$0 < 1 \quad (\text{non-triviality})$$

# Linear Rational Arithmetic

---

Note: The second non-triviality axiom renders the first one superfluous. Moreover, as soon as we add the axioms of compatible strict total orderings, torsion-freeness can be omitted. Every ordered divisible abelian group is obviously torsion-free.

In fact the converse holds: Every torsion-free abelian group can be ordered (F.-W. Levi 1913).

Examples:  $\mathbb{Q}$ ,  $\mathbb{R}$ ,  $\mathbb{Q}^n$ ,  $\mathbb{R}^n$ , ...

# Linear Rational Arithmetic

---

The signature can be extended by further symbols:

$\leq/2, >/2, \geq/2, \neq/2$ : defined using  $<$  and  $\approx$

$-/1$ : Skolem function for inverse axiom

$-/2$ : defined using  $+/2$  and  $-/1$

$\text{div}_n/1$ : Skolem functions for divisibility axiom for all  $n \geq 1$ .

$\text{mult}_n/1$ : defined by  $\forall x (\text{mult}_n(x) \approx \underbrace{x + \cdots + x}_{n \text{ times}})$  for all  $n \geq 1$ .

$\text{mult}_q/1$ : defined using  $\text{mult}_n, \text{div}_n, -$  for all  $q \in \mathbb{Q}$ .

(We usually write  $q \cdot t$  or  $qt$  instead of  $\text{mult}_q(t)$ .)

$q/0$  (for  $q \in \mathbb{Q}$ ): defined by  $q \approx q \cdot 1$ .

# Linear Rational Arithmetic

---

Note: Every formula using the additional symbols is ODAG-equivalent to a formula over the base signature.

When  $\cdot$  is considered as a binary operator, (ordered) divisible torsion-free abelian groups correspond to (ordered) rational vector spaces.

# Fourier-Motzkin Quantifier Elimination

---

Linear rational arithmetic permits **quantifier elimination**: every formula  $\exists x F$  or  $\forall x F$  in linear rational arithmetic can be converted into an equivalent formula without the variable  $x$ .

The method was discovered in 1826 by J. Fourier and re-discovered by T. Motzkin in 1936.

# Fourier-Motzkin Quantifier Elimination

---

Observation: Every literal over the variables  $x, y_1, \dots, y_n$  can be converted into an ODAG-equivalent literal  $x \sim t[\vec{y}]$  or  $0 \sim t[\vec{y}]$ , where  $\sim \in \{<, >, \leq, \geq, \approx, \neq\}$  and  $t[\vec{y}]$  has the form  $\sum_i q_i \cdot y_i + q_0$ .

In other words, we can either eliminate  $x$  completely or isolate it on one side of the literal, and we can replace every negative ordering literal by a positive one.

Moreover, we can convert every  $\neq$ -literal into an ODAG-equivalent disjunction of two  $<$ -literals.

# Fourier-Motzkin Quantifier Elimination

---

We first consider existentially quantified conjunctions of atoms.

If the conjunction contains an equation  $x \approx t[\vec{y}]$ , we can eliminate the quantifier  $\exists x$  by substitution:

$$\exists x (x \approx t[\vec{y}] \wedge F)$$

is equivalent to

$$F \{x \mapsto t[\vec{y}]\}$$

# Fourier-Motzkin Quantifier Elimination

---

If  $x$  occurs only in inequations, then

$$\begin{aligned} \exists x \left( \bigwedge_i x < s_i(\vec{y}) \wedge \bigwedge_j x \leq t_j(\vec{y}) \right. \\ \left. \wedge \bigwedge_k x > u_k(\vec{y}) \wedge \bigwedge_l x \geq v_l(\vec{y}) \wedge \bigwedge_m 0 \sim_m w_m(\vec{y}) \right) \end{aligned}$$

is equivalent to

$$\begin{aligned} \bigwedge_i \bigwedge_k s_i(\vec{y}) > u_k(\vec{y}) \wedge \bigwedge_j \bigwedge_k t_j(\vec{y}) > u_k(\vec{y}) \\ \wedge \bigwedge_i \bigwedge_l s_i(\vec{y}) > v_l(\vec{y}) \wedge \bigwedge_j \bigwedge_l t_j(\vec{y}) \geq v_l(\vec{y}) \\ \wedge \bigwedge_m 0 \sim_m w_m(\vec{y}) \end{aligned}$$

Proof: ( $\Rightarrow$ ) by transitivity;

( $\Leftarrow$ ) take  $\frac{1}{2}(\min\{s_i, t_j\} + \max\{u_k, v_l\})$  as a witness.

# Fourier-Motzkin Quantifier Elimination

---

Extension to arbitrary formulas:

Transform into prenex formula;

if innermost quantifier is  $\exists$ : transform matrix into DNF and move  $\exists$  into disjunction;

if innermost quantifier is  $\forall$ : replace  $\forall x F$  by  $\neg \exists x \neg F$ , then eliminate  $\exists$ .

# Fourier-Motzkin Quantifier Elimination

---

Consequence: every closed formula over the signature of ODAGs is ODAG-equivalent to either  $\top$  or  $\perp$ .

Consequence: ODAGs are a **complete** theory, i. e., every closed formula over the signature of ODAGs is either valid or unsatisfiable w. r. t. ODAGs.

# Fourier-Motzkin Quantifier Elimination

---

Consequence: every closed formula over the signature of ODAGs holds either in all ODAGs or in no ODAG.

ODAGs are indistinguishable by first-order formulas over the signature of ODAGs.

(These properties do not hold for extended signatures!)

# Fourier-Motzkin: Complexity

---

One FM-step for  $\exists$ :

formula size grows quadratically, therefore  $O(n^2)$  runtime.

$m$  quantifiers  $\exists \dots \exists$ :

naive implementation produces a doubly exponential number of inequations, therefore needs  $O(n^{2^m})$  runtime  
(the number of *necessary* inequations grows only exponentially, though).

$m$  quantifiers  $\exists \forall \exists \forall \dots \exists$ :

CNF/DNF conversion (exponential!) required after each step;  
therefore non-elementary runtime.

# Loos-Weispfenning Quantifier Elimination

---

A more efficient way to eliminate quantifiers in linear rational arithmetic was developed by R. Loos and V. Weispfenning (1993).

The method is also known as “test point method” or “virtual substitution method”.

# Loos-Weispfenning Quantifier Elimination

---

For simplicity, we consider only one particular ODAG, namely  $\mathbb{Q}$  (as we have seen above, the results are the same for all ODAGs).

# Loos-Weispfenning Quantifier Elimination

---

Let  $F(x, \vec{y})$  be a *positive* boolean combination of linear (in-)equations  $x \sim_i s_i(\vec{y})$  and  $0 \sim_j s'_j(\vec{y})$  with  $\sim_i, \sim_j \in \{\approx, \neq, <, \leq, >, \geq\}$ , that is, a formula built from linear (in-)equations,  $\wedge$  and  $\vee$  (but without  $\neg$ ).

Goal: Find a *finite* set  $T$  of “test points” so that

$$\exists x F(x, \vec{y}) \quad \models \quad \bigvee_{t \in T} F(x, \vec{y}) \{x \mapsto t\}$$

In other words: We want to replace the infinite disjunction  $\exists x$  by a finite disjunction.

# Loos-Weispfenning Quantifier Elimination

---

If we keep the values of the variables  $\vec{y}$  fixed, then we can consider  $F$  as a function  $F : x \mapsto F(x, \vec{y})$  from  $\mathbb{Q}$  to  $\{0, 1\}$ .

The value of each of the atoms  $s_i(\vec{y}) \sim_i x$  changes only at  $s_i(\vec{y})$ , and the value of  $F$  can only change if the value of one of its atoms changes.

# Loos-Weispfenning Quantifier Elimination

---

Let  $\delta(\vec{y}) = \min\{ |s_i(\vec{y}) - s_j(\vec{y})| \mid s_i(\vec{y}) \neq s_j(\vec{y}) \}$

$F$  is a piecewise constant function; more precisely, the set of all  $x$  with  $F(x, \vec{y}) = 1$  is a finite union of intervals. (The union may be empty, the individual intervals may be finite or infinite and open or closed.)

Moreover, each of the intervals has either length 0 (i. e., it consists of one point), or its length is at least  $\delta(\vec{y})$ .

# Loos-Weispfenning Quantifier Elimination

---

If the set of all  $x$  for which  $F(x, \vec{y})$  is 1 is non-empty, then

- (i)  $F(x, \vec{y}) = 1$  for all  $x \leq r(\vec{y})$  for some  $r(\vec{y}) \in \mathbb{Q}$
- (ii) or there is some point where the value of  $F(x, \vec{y})$  switches from 0 to 1 when we traverse the real axis from  $-\infty$  to  $+\infty$ .

We use this observation to construct a set of test points.

We start with some “sufficiently small” test point  $r(\vec{y})$  to take care of case (i).

# Loos-Weispfenning Quantifier Elimination

---

For case (ii), we observe that  $F(x, \vec{y})$  can only switch from 0 to 1 if one of the atoms switches from 0 to 1. (We consider only *positive* boolean combinations of atoms, and  $\wedge$  and  $\vee$  are monotonic w. r. t. truth values.)

$x \leq s_i(\vec{y})$  and  $x < s_i(\vec{y})$  do not switch from 0 to 1 when  $x$  grows.

$x \geq s_i(\vec{y})$  and  $x \approx s_i(\vec{y})$  switch from 0 to 1 at  $s_i(\vec{y})$

$\Rightarrow s_i(\vec{y})$  is a test point.

$x > s_i(\vec{y})$  and  $x \not\approx s_i(\vec{y})$  switch from 0 to 1 “right after”  $s_i(\vec{y})$

$\Rightarrow s_i(\vec{y}) + \varepsilon$  (for some  $0 < \varepsilon < \delta(\vec{y})$ ) is a test point.

# Loos-Weispfenning Quantifier Elimination

---

If  $r(\vec{y})$  is sufficiently small and  $0 < \varepsilon < \delta(\vec{y})$ , then

$$T := \{r(\vec{y})\} \cup \{s_i(\vec{y}) \mid \sim_i \in \{\geq, =\}\} \\ \cup \{s_i(\vec{y}) + \varepsilon \mid \sim_i \in \{>, \neq\}\}.$$

is a set of test points.

Problem:

We don't know how small  $r(\vec{y})$  has to be for case (i), and we don't know  $\delta(\vec{y})$  for case (ii).

# Loos-Weispfenning Quantifier Elimination

---

Idea:

We consider the limits for  $r \rightarrow -\infty$  and for  $\varepsilon \searrow 0$ , that is, we redefine

$$\begin{aligned} T := & \{-\infty\} \cup \{s_i(\vec{y}) \mid \sim_i \in \{\geq, =\}\} \\ & \cup \{s_i(\vec{y}) + \varepsilon \mid \sim_i \in \{>, \neq\}\}. \end{aligned}$$

How can we eliminate the infinitesimals  $\infty$  and  $\varepsilon$  when we substitute elements of  $T$  for  $x$ ?

# Loos-Weispfenning Quantifier Elimination

---

Virtual substitution:

$$(x < s(\vec{y})) \{x \mapsto -\infty\} := \lim_{r \rightarrow -\infty} (r < s(\vec{y})) = \top$$

$$(x \leq s(\vec{y})) \{x \mapsto -\infty\} := \lim_{r \rightarrow -\infty} (r \leq s(\vec{y})) = \top$$

$$(x > s(\vec{y})) \{x \mapsto -\infty\} := \lim_{r \rightarrow -\infty} (r > s(\vec{y})) = \perp$$

$$(x \geq s(\vec{y})) \{x \mapsto -\infty\} := \lim_{r \rightarrow -\infty} (r \geq s(\vec{y})) = \perp$$

$$(x \approx s(\vec{y})) \{x \mapsto -\infty\} := \lim_{r \rightarrow -\infty} (r \approx s(\vec{y})) = \perp$$

$$(x \not\approx s(\vec{y})) \{x \mapsto -\infty\} := \lim_{r \rightarrow -\infty} (r \not\approx s(\vec{y})) = \top$$

# Loos-Weispfenning Quantifier Elimination

---

Virtual substitution:

$$(x < s(\vec{y})) \{x \mapsto u + \varepsilon\} := \lim_{\varepsilon \searrow 0} (u + \varepsilon < s(\vec{y})) = (u < s(\vec{y}))$$

$$(x \leq s(\vec{y})) \{x \mapsto u + \varepsilon\} := \lim_{\varepsilon \searrow 0} (u + \varepsilon \leq s(\vec{y})) = (u < s(\vec{y}))$$

$$(x > s(\vec{y})) \{x \mapsto u + \varepsilon\} := \lim_{\varepsilon \searrow 0} (u + \varepsilon > s(\vec{y})) = (u \geq s(\vec{y}))$$

$$(x \geq s(\vec{y})) \{x \mapsto u + \varepsilon\} := \lim_{\varepsilon \searrow 0} (u + \varepsilon \geq s(\vec{y})) = (u \geq s(\vec{y}))$$

$$(x \approx s(\vec{y})) \{x \mapsto u + \varepsilon\} := \lim_{\varepsilon \searrow 0} (u + \varepsilon \approx s(\vec{y})) = \perp$$

$$(x \not\approx s(\vec{y})) \{x \mapsto u + \varepsilon\} := \lim_{\varepsilon \searrow 0} (u + \varepsilon \not\approx s(\vec{y})) = \top$$

# Loos-Weispfenning Quantifier Elimination

---

We have traversed the real axis from  $-\infty$  to  $+\infty$ . Alternatively, we can traverse it from  $+\infty$  to  $-\infty$ . In this case, the test points are

$$\begin{aligned} T' := & \{+\infty\} \cup \{s_i(\vec{y}) \mid \sim_i \in \{\leq, =\}\} \\ & \cup \{s_i(\vec{y}) - \varepsilon \mid \sim_i \in \{<, \neq\}\}. \end{aligned}$$

Infinitesimals are eliminated in a similar way as before.

In practice: Compute both  $T$  and  $T'$  and take the smaller set.

# Loos-Weispfenning Quantifier Elimination

---

For a universally quantified formulas  $\forall x F$ , we replace it by  $\neg \exists x \neg F$ , push inner negation downwards, and then continue as before.

Note that there is no CNF/DNF transformation required. Loos-Weispfenning quantifier elimination works on arbitrary positive formulas.

# Loos-Weispfenning: Complexity

---

One LW-step for  $\exists$  or  $\forall$ :

as the number of test points is at most one plus the number of atoms (one plus half of the number of atoms, if there are only ordering literals), the formula size grows quadratically; therefore  $O(n^2)$  runtime.

# Loos-Weispfenning: Complexity

---

Multiple quantifiers of the same kind:

$$\exists x_2 \exists x_1. F(x_1, x_2, \vec{y})$$

$$\rightsquigarrow \exists x_2. \left( \bigvee_{t_1 \in T_1} F(x_1, x_2, \vec{y}) \{x_1 \mapsto t_1\} \right)$$

$$\rightsquigarrow \bigvee_{t_1 \in T_1} \left( \exists x_2. F(x_1, x_2, \vec{y}) \{x_1 \mapsto t_1\} \right)$$

$$\rightsquigarrow \bigvee_{t_1 \in T_1} \bigvee_{t_2 \in T_2} \left( F(x_1, x_2, \vec{y}) \{x_1 \mapsto t_1\} \{x_2 \mapsto t_2\} \right)$$

# Loos-Weispfenning: Complexity

---

$m$  quantifiers  $\exists \dots \exists$  or  $\forall \dots \forall$ :

formula size is multiplied by  $n$  in each step, therefore  $O(n^{m+1})$  runtime.

$m$  quantifiers  $\exists \forall \exists \forall \dots \exists$ :

doubly exponential runtime.

Note: The formula resulting from a LW-step is usually highly redundant; so an efficient implementation must make heavy use of simplification techniques.

## 1.4 Existentially-quantified LRA

---

So far, we have considered formulas that may contain free, existentially quantified, and universally quantified variables.

For the special case of conjunction of linear inequations in which *all* variables are existentially quantified, there are more efficient methods available.

Main idea: reduce satisfiability problem to optimization problem.

# Linear Optimization

---

Goal:

Solve a linear optimization (also called: linear programming) problem for given numbers  $a_{ij}, b_i, c_j \in \mathbb{R}$ :

$$\begin{aligned} &\text{maximize } \sum_{1 \leq j \leq n} c_j x_j \\ &\text{for } \bigwedge_{1 \leq i \leq m} \sum_{1 \leq j \leq n} a_{ij} x_j \leq b_i \end{aligned}$$

or in vectorial notation:

$$\begin{aligned} &\text{maximize } \vec{c}^\top \vec{x} \\ &\text{for } A\vec{x} \leq \vec{b} \end{aligned}$$

# Linear Optimization

---

Simplex algorithm:

Developed independently by Kantorovich (1939), Dantzig (1948).

Polynomial-time average-case complexity; worst-case time complexity is exponential, though.

Interior point methods:

First algorithm by Karmarkar (1984).

Polynomial-time worst-case complexity (but large constants).

In practice: no clear winner.

# Linear Optimization

---

Implementations:

GLPK (GNU Linear Programming Kit),

Gurobi.

# Linear Optimization

---

Main idea of Simplex:

$A\vec{x} \leq \vec{b}$  describes a convex polyhedron.

Pick one vertex of the polyhedron,  
then follow the edges of the polyhedron towards an optimal solution.

By convexity, the local optimum found in this way is also a global optimum.

Details: see special lecture on optimization.

# Linear Optimization

---

Using an optimization procedure for checking satisfiability:

Goal: Check whether  $A\vec{x} \leq \vec{b}$  is satisfiable.

To use the Simplex method, we have to transform the original (possibly empty) polyhedron into another polyhedron that is non-empty and for which we know one initial vertex.

Every real number can be written as the difference of two non-negative real numbers.

Use this idea to convert  $A\vec{x} \leq \vec{b}$  into an equisatisfiable inequation system  $\vec{y} \geq \vec{0}$ ,  $B\vec{y} \leq \vec{b}$  for new variables  $\vec{y}$ .

# Linear Optimization

---

Multiply those inequations of the inequation system  $B\vec{y} \leq \vec{b}$  in which the number on the right-hand side is negative by  $-1$ . We obtain two inequation systems  $D_1\vec{y} \leq \vec{g}_1$ ,  $D_2\vec{y} \geq \vec{g}_2$ , such that  $\vec{g}_1 \geq \vec{0}$ ,  $\vec{g}_2 > 0$ .

Now solve

$$\text{maximize } \vec{1}^\top (D_2\vec{y} - \vec{z})$$

$$\text{for } \vec{y}, \vec{z} \geq \vec{0}$$

$$D_1\vec{y} \leq \vec{g}_1$$

$$D_2\vec{y} - \vec{z} \leq \vec{g}_2$$

where  $\vec{z}$  is a vector of new variables with the same size as  $\vec{g}_2$ .

# Linear Optimization

---

Observation 1:

$\vec{0}$  is a vertex of the polyhedron of this optimization problem.

Observation 2:

The maximum is  $\vec{1}^\top \vec{g}_2$  if and only if  $\vec{y} \geq \vec{0}$ ,  $D_1 \vec{y} \leq \vec{g}_1$ ,  $D_2 \vec{y} \geq \vec{g}_2$  has a solution.

( $\Rightarrow$ ): If  $\vec{1}^\top (D_2 \vec{y} - \vec{z}) = \vec{1}^\top \vec{g}_2$  for some  $\vec{y}, \vec{z}$  satisfying  $D_2 \vec{y} - \vec{z} \leq \vec{g}_2$ , then  $D_2 \vec{y} - \vec{z} = \vec{g}_2$ , hence  $D_2 \vec{y} = \vec{g}_2 + \vec{z} \geq \vec{g}_2$ .

( $\Leftarrow$ ):  $\vec{1}^\top (D_2 \vec{y} - \vec{z})$  can never be larger than  $\vec{1}^\top \vec{g}_2$ . If  $\vec{y} \geq \vec{0}$ ,  $D_1 \vec{y} \leq \vec{g}_1$ ,  $D_2 \vec{y} \geq \vec{g}_2$  has a solution, choose  $\vec{z} = D_2 \vec{y} - \vec{g}_2$ ; then  $\vec{1}^\top (D_2 \vec{y} - \vec{z}) = \vec{1}^\top \vec{g}_2$ .

# Linear Optimization

---

A Simplex variant:

Transform the satisfiability problem into the form

$$A\vec{x} = \vec{0}$$
$$\vec{l} \leq \vec{x} \leq \vec{u}$$

(where  $l_i$  may be  $-\infty$  and  $u_i$  may be  $+\infty$ ).

Relation to optimization problem is obscured.

But: More efficient if one needs an incremental decision procedure, where inequations may be added and retracted (Dutertre and de Moura 2006).

## 1.5 Non-linear Real Arithmetic

---

Tarski (1951): Quantifier elimination is possible for *non-linear* real arithmetic (or more generally, for real-closed fields).

His algorithm had non-elementary complexity, however.

An improved algorithm by Collins (1975) (with further improvements by Hong) has doubly exponential complexity: Cylindrical algebraic decomposition (CAD).

Implementation: QEPCAD.

# Cylindrical Algebraic Decomposition

---

Given: First-order formula over atoms of the form  $f_i(\vec{x}) \sim 0$ , where the  $f_i$  are polynomials over variables  $\vec{x}$ .

Goal: Decompose  $\mathbb{R}^n$  into a finite number of regions such that all polynomials have invariant sign on every region  $X$ :

$$\begin{aligned} \forall i \ ( \forall \vec{x} \in X. f_i(\vec{x}) < 0 \\ \vee \forall \vec{x} \in X. f_i(\vec{x}) = 0 \\ \vee \forall \vec{x} \in X. f_i(\vec{x}) > 0 ) \end{aligned}$$

Note: Implementation needs exact arithmetic using algebraic numbers (i. e., zeroes of univariate polynomials with integer coefficients).

## 1.6 Real Arithmetic incl. Transcendental Fctns.

---

Real arithmetic with exp/log: decidability unknown.

Real arithmetic with trigonometric functions: undecidable

The following formula holds exactly if  $x \in \mathbb{Z}$ :

$$\exists y (\sin(y) = 0 \wedge 3 < y \wedge y < 4 \wedge \sin(x \cdot y) = 0)$$

(note that necessarily  $y = \pi$ ).

Consequence: Peano arithmetic (which is undecidable) can be encoded in real arithmetic with trigonometric functions.

## Real Arithmetic incl. Transcendental Fctns.

---

However, real arithmetic with transcendental functions is decidable for formulas that are *stable under perturbations*, i. e., whose truth value does not change if numeric constants are modified by some sufficiently small  $\varepsilon$ .

Example:

Stable under perturbations:  $\exists x \ x^2 \leq 5$

Not stable under perturbations:  $\exists x \ x^2 \leq 0$

(Formula is true, but if we subtract an arbitrarily small  $\varepsilon > 0$  from the right-hand side, it becomes false.)

## Real Arithmetic incl. Transcendental Fctns.

Unsatisfactory from a mathematical point of view, but sufficient for engineering applications (where stability under perturbations is necessary anyhow).

Approach:

Interval arithmetic + interval bisection if necessary (Ratschan).

Sound for general formulas; complete for formulas that are stable under perturbations; may loop forever if the formula is not stable under perturbations.

## 1.7 Linear Integer Arithmetic

---

Linear integer arithmetic = Presburger arithmetic.

Decidable (Presburger, 1929), but quantifier elimination is only possible if additional divisibility operators are present:

$\exists x (y = 2x)$  is equivalent to  $\text{divides}(2, y)$  but not to any quantifier-free formula over the base signature.

Cooper (1972): Quantifier elimination procedure, triple exponential for arbitrarily quantified formulas.

# The Omega Test

---

Omega test (Pugh, 1991): variant of Fourier–Motzkin for conjunctions of (in-)equations in linear integer arithmetic.

Idea:

- Perform easy transformations, e. g.:

$$3x + 6y \leq 8 \mapsto 3x + 6y \leq 6 \mapsto x + 2y \leq 2$$

$$3x + 6y = 8 \mapsto \perp$$

(since  $3x + 6y$  must be divisible by 3).

- Eliminate equations

(easy, if one coefficient is 1; tricky otherwise).

# The Omega Test

---

- If only inequations are left:
  - no real solutions  $\rightarrow$  unsatisfiable for  $\mathbb{Z}$
  - “sufficiently many” real solutions  $\rightarrow$  satisfiable for  $\mathbb{Z}$
  - otherwise: branch

# The Omega Test

---

What does “sufficiently many” mean?

Consider inequations  $ax \leq s$  and  $bx \geq t$  with  $a, b \in \mathbb{N}^{>0}$  and polynomials  $s, t$ .

If these inequations have real solutions, the interval of solutions ranges from  $\frac{1}{b}t$  to  $\frac{1}{a}s$ .

The longest possible interval of this kind that does not contain any integer number ranges from  $i + \frac{1}{b}$  to  $i + 1 - \frac{1}{a}$  for some  $i \in \mathbb{Z}$ ; it has the length  $1 - \frac{1}{a} - \frac{1}{b}$ .

# The Omega Test

---

Consequence:

If  $\frac{1}{a}s > \frac{1}{b}t + (1 - \frac{1}{a} - \frac{1}{b})$ , or equivalently,  $bs \geq at + ab - a - b + 1$  is satisfiable, then the original problem must have integer solutions.

It remains to consider the case that  $bs \geq at$  is satisfiable (hence there are real solutions) but  $bs \geq at + ab - a - b + 1$  is not (hence the interval of real solutions need not contain an integer).

# The Omega Test

---

In the latter case,  $bs \leq at + ab - a - b$  holds, hence for every solution of the original problem:

$$t \leq bx \leq \frac{b}{a}s \leq t + (b - 1 - \frac{b}{a})$$

and if  $x$  is an integer,  $t \leq bx \leq t + \lfloor b - 1 - \frac{b}{a} \rfloor$

⇒ Branch non-deterministically:

Add one of the equations  $bx = t + i$

for  $i \in \{0, \dots, \lfloor b - 1 - \frac{b}{a} \rfloor\}$ .

Alternatively, if  $b > a$ :

Add one of the equations  $ax = s - i$

for  $i \in \{0, \dots, \lfloor a - 1 - \frac{a}{b} \rfloor\}$ .

# The Omega Test

---

Note: Efficiency depends highly on the size of coefficients.

In applications from program verification, there is almost always some variable with a very small coefficient.

If all coefficients are large, the branching step gets expensive.

# Branch-and-Cut

---

Alternative approach: Reduce satisfiability problem to optimization problem (like Simplex).

ILP, MILP: (mixed) integer linear programming.

# Branch-and-Cut

---

Two basic approaches:

Branching:

If the simplex algorithm finds a solution with  $x = 2.7$ , add the inequation  $x \leq 2$  or the inequation  $x \geq 3$ .

Cutting planes:

Derive an inequation that holds for all real solutions, then round it to obtain an inequation that holds for all integer solutions, but not for the real solution found previously.

# Branch-and-Cut

---

Example:

$$\text{Given: } 2x - 3y \leq 1$$

$$2x + 3y \leq 5$$

$$-5x - 4y \leq -7$$

Simplex finds an extremal solution  $x = \frac{3}{2}$ ,  $y = \frac{2}{3}$ .

From the first two inequations, we see that  $4x \leq 6$ ,  
hence  $x \leq \frac{3}{2}$ . If  $x \in \mathbb{Z}$ , we conclude  $x = \lfloor x \rfloor \leq \lfloor \frac{3}{2} \rfloor = 1$ .

$\Rightarrow$  Add the inequation  $x \leq 1$ , which holds for all integer solutions, but cuts off the solution  $(\frac{3}{2}, \frac{2}{3})$ .

# Branch-and-Cut

---

In practice:

Use both: Alternate between branching and cutting steps.

Better performance than the individual approaches.

## 1.8 Difference Logic

---

Difference Logic (DL):

Fragment of linear rational or integer arithmetic.

Formulas: conjunctions of atoms  $x - y < c$  or  $x - y \leq c$ ,  
 $x, y \in X$ ,  $c \in \mathbb{Q}$  (or  $c \in \mathbb{Z}$ ).

One special variable  $x_0$  whose value is fixed to 0 is permitted;  
this allows to express atoms like  $x < 3$  in the form  $x - x_0 < 3$ .

# Difference Logic

---

Solving difference logic:

Let  $F$  be a conjunction in DL.

For simplicity: only non-strict inequalities.

Define a weighted graph  $G$ :

Vertices  $V$ : Variables in  $F$ .

Edges  $E$ :  $x - y \leq c \rightsquigarrow$  edge  $(x, y)$  with weight  $c$ .

Theorem:  $F$  is unsatisfiable iff  $G$  has a negative cycle.

Can be checked in  $O(|V| \cdot |E|)$  using the Bellman-Ford algorithm.

## 1.9 C-Arithmetic

---

In languages like C: Bounded integer arithmetic (modulo  $2^n$ ),  
in device drivers also combined with bitwise operations.

Bit-Blasting (encode everything as boolean circuits, use CDCL):

Naive encoding: possible, but often too inefficient.

If combined with over-/underapproximation techniques (Bryant, Kroening, et al.): successful.

## 1.10 Decision Procedures for Data Structures

---

There are decision procedures for, e. g.,

Arrays (read, write)

Lists (car, cdr, cons)

Sets or multisets with cardinalities

Bitvectors

Note: There are usually restrictions on quantifications. Unrestricted universal quantification can lead to undecidability.

## 1.11 Combining Decision Procedures

---

Problem:

Let  $\mathcal{T}_1$  and  $\mathcal{T}_2$  be first-order theories over the signatures  $\Sigma_1$  and  $\Sigma_2$ .

Assume that we have decision procedures for the satisfiability of existentially quantified formulas (or the validity of universally quantified formulas) w. r. t.  $\mathcal{T}_1$  and  $\mathcal{T}_2$ .

Can we combine them to get a decision procedure for the satisfiability of existentially quantified formulas w. r. t.  $\mathcal{T}_1 \cup \mathcal{T}_2$ ?

# Combining Decision Procedures

---

General assumption:

$\Sigma_1$  and  $\Sigma_2$  are disjoint.

The only symbol shared by  $\mathcal{T}_1$  and  $\mathcal{T}_2$  is built-in equality.

We consider only conjunctions of literals.

For general formulas, convert to DNF first and consider each conjunction individually.

# Abstraction

---

To be able to use the individual decision procedures, we have to transform the original formula in such a way that each atom contains only symbols of one of the signatures (plus variables).

This process is known as **variable abstraction** or **purification**.

# Abstraction

---

We apply the following rule as long as possible:

$$\frac{\exists \vec{x} (F[t])}{\exists \vec{x}, y (F[y] \wedge t \approx y)}$$

if the top symbol of  $t$  belongs to  $\Sigma_i$  and  $t$  occurs in  $F$  directly below a  $\Sigma_j$ -symbol or in a (positive or negative) equation  $s \approx t$  where the top symbol of  $s$  belongs to  $\Sigma_j$  ( $i \neq j$ ), and if  $y$  is a new variable.

It is easy to see that the original and the purified formula are equivalent.

# Stable Infiniteness

---

Problem:

Even if the  $\Sigma_1$ -formula  $F_1$  and the  $\Sigma_2$ -formula  $F_2$  do not share any symbols (not even variables), and if  $F_1$  is  $\mathcal{T}_1$ -satisfiable and  $F_2$  is  $\mathcal{T}_2$ -satisfiable, we cannot conclude that  $F_1 \wedge F_2$  is  $(\mathcal{T}_1 \cup \mathcal{T}_2)$ -satisfiable.

# Stable Infiniteness

---

Example:

Consider

$$\mathcal{T}_1 = \{\forall x, y, z (x \approx y \vee x \approx z \vee y \approx z)\}$$

and

$$\mathcal{T}_2 = \{\exists x, y, z (x \not\approx y \wedge x \not\approx z \wedge y \not\approx z)\}.$$

All  $\mathcal{T}_1$ -models have at most two elements, and all  $\mathcal{T}_2$ -models have at least three elements.

Since  $\mathcal{T}_1 \cup \mathcal{T}_2$  is contradictory, there are no  $(\mathcal{T}_1 \cup \mathcal{T}_2)$ -satisfiable formulas.

# Stable Infiniteness

---

To ensure that  $\mathcal{T}_1$ -models and  $\mathcal{T}_2$ -models can be combined to  $(\mathcal{T}_1 \cup \mathcal{T}_2)$ -models, we require that both  $\mathcal{T}_1$  and  $\mathcal{T}_2$  are stably infinite.

A first-order theory  $\mathcal{T}$  is called **stably infinite**, if every existentially quantified formula that has a  $\mathcal{T}$ -model has also a  $\mathcal{T}$ -model with a (countably) infinite universe.

Note: By the Löwenheim–Skolem theorem, “countable” is redundant here.

## Shared Variables

---

Even if  $\exists \vec{x} F_1$  is  $\mathcal{T}_1$ -satisfiable and  $\exists \vec{x} F_2$  is  $\mathcal{T}_2$ -satisfiable, it can happen that  $\exists \vec{x} (F_1 \wedge F_2)$  is not  $(\mathcal{T}_1 \cup \mathcal{T}_2)$ -satisfiable, for instance because the shared variables  $x$  and  $y$  must be equal in all  $\mathcal{T}_1$ -models of  $\exists \vec{x} F_1$  and different in all  $\mathcal{T}_2$ -models of  $\exists \vec{x} F_2$ .

# Shared Variables

---

Example:

Consider

$$F_1 = (x + (-y) \approx 0),$$

and

$$F_2 = (f(x) \neq f(y))$$

where  $\mathcal{T}_1$  is linear rational arithmetic and  $\mathcal{T}_2$  is EUF.

We must exchange information about shared variables to detect the contradiction.

## The Nelson–Oppen Algorithm (Non-determ.)

Suppose that  $\exists \vec{x} F$  is a purified conjunction of  $\Sigma_1$  and  $\Sigma_2$ -literals.

Let  $F_1$  be the conjunction of all literals of  $F$  that do not contain  $\Sigma_2$ -symbols; let  $F_2$  be the conjunction of all literals of  $F$  that do not contain  $\Sigma_1$ -symbols. (Equations between variables are in both  $F_1$  and  $F_2$ .)

The Nelson–Oppen algorithm starts with the pair  $F_1, F_2$  and applies the following inference rules.

# The Nelson–Oppen Algorithm (Non-determ.)

---

Unsat:

$$\frac{F_1, F_2}{\perp}$$

if  $\exists \vec{x} F_i$  is unsatisfiable w. r. t.  $\mathcal{T}_i$  for some  $i$ .

Branch:

$$\frac{F_1, F_2}{F_1 \wedge (x \approx y), F_2 \wedge (x \approx y) \quad | \quad F_1 \wedge (x \not\approx y), F_2 \wedge (x \not\approx y)}$$

if  $x$  and  $y$  are two different variables appearing in both  $F_1$  and  $F_2$  such that neither  $x \approx y$  nor  $x \not\approx y$  occurs in both  $F_1$  and  $F_2$

## The Nelson–Oppen Algorithm (Non-determ.)

---

“|” means non-deterministic (backtracking!) branching of the derivation into two subderivations. Derivations are therefore trees. All branches need to be reduced until termination.

Clearly, all derivation paths are finite since there are only finitely many **shared variables** in  $F_1$  and  $F_2$ , therefore the procedure represented by the rules is terminating.

We call a constraint configuration to which no rule applies **irreducible**.

# The Nelson–Oppen Algorithm (Non-determ.)

---

Theorem 1.2 (Soundness):

If “Branch” can be applied to  $F_1, F_2$ , then  $\exists \vec{x} (F_1 \wedge F_2)$  is satisfiable in  $\mathcal{T}_1 \cup \mathcal{T}_2$  if and only if one of the successor configurations of  $F_1, F_2$  is satisfiable in  $\mathcal{T}_1 \cup \mathcal{T}_2$ .

Corollary 1.3:

If all paths in a derivation tree from  $F_1, F_2$  end in  $\perp$ , then  $\exists \vec{x} (F_1 \wedge F_2)$  is unsatisfiable in  $\mathcal{T}_1 \cup \mathcal{T}_2$ .

## The Nelson–Oppen Algorithm (Non-determ.)

---

For completeness we need to show that if one branch in a derivation terminates with an irreducible configuration  $F_1, F_2$  (different from  $\perp$ ), then  $\exists \vec{x} (F_1 \wedge F_2)$  (and, thus, the initial formula of the derivation) is satisfiable in the combined theory.

As  $\exists \vec{x} (F_1 \wedge F_2)$  is irreducible by “Unsat”, the two formulas are satisfiable in their respective component theories, that is, we have  $\mathcal{T}_i$ -models  $\mathcal{A}_i$  of  $\exists \vec{x} F_i$  for  $i \in \{1, 2\}$ . We are left with combining the models into a single one that is both a model of the combined theory and of the combined formula. These constructions are called **amalgamations**.

## The Nelson–Oppen Algorithm (Non-determ.)

---

Let  $F$  be a  $\Sigma_i$ -formula and let  $S$  be a set of variables of  $F$ .

$F$  is called **compatible** with an equivalence  $\sim$  on  $S$  if the formula

$$\exists \vec{z} \left( F \wedge \bigwedge_{x,y \in S, x \sim y} x \approx y \wedge \bigwedge_{x,y \in S, x \not\sim y} x \not\approx y \right) \quad (1)$$

is  $\mathcal{T}_i$ -satisfiable whenever  $F$  is  $\mathcal{T}_i$ -satisfiable.

This expresses that  $F$  does not contradict equalities between the variables in  $S$  as given by  $\sim$ .

The formula  $\bigwedge_{x,y \in S, x \sim y} x \approx y \wedge \bigwedge_{x,y \in S, x \not\sim y} x \not\approx y$  is called an *arrangement* of  $S$ .

## The Nelson–Oppen Algorithm (Non-determ.)

---

Proposition 1.4:

If  $F_1, F_2$  is a pair of conjunctions over  $\mathcal{T}_1$  and  $\mathcal{T}_2$ , respectively, that is irreducible by “Branch”, then both  $F_1$  and  $F_2$  are compatible with some equivalence  $\sim$  on the shared variables  $S$  of  $F_1$  and  $F_2$ .

Proof:

If  $F_1, F_2$  is irreducible by the branching rule, then for each pair of shared variables  $x$  and  $y$ , both  $F_1$  and  $F_2$  contain either  $x \approx y$  or  $x \not\approx y$ .

Choose  $\sim$  to be the equivalence given by all (positive) variable equations between shared variables that are contained in  $F_1$ .

## The Nelson–Oppen Algorithm (Non-determ.)

---

Let  $\Sigma = (\Omega, \Pi)$ ; let  $\Sigma' = (\Omega', \Pi')$  with  $\Omega' \subseteq \Omega$  and  $\Pi' \subseteq \Pi$  be a subsignature of  $\Sigma$ .

Let  $\mathcal{A}$  be a  $\Sigma$ -algebra. Then the **reduct**  $\mathcal{A}|_{\Sigma'}$  is the  $\Sigma'$ -algebra  $\mathcal{A}'$  with

$$U_{\mathcal{A}'} = U_{\mathcal{A}},$$

$$f_{\mathcal{A}'} = f_{\mathcal{A}} \text{ for all } f \in \Omega', \text{ and}$$

$$P_{\mathcal{A}'} = P_{\mathcal{A}} \text{ for all } P \in \Pi'.$$

## The Nelson–Oppen Algorithm (Non-determ.)

---

Lemma 1.5 (Amalgamation Lemma):

Let  $\mathcal{T}_1$  and  $\mathcal{T}_2$  be two stably infinite theories over disjoint signatures  $\Sigma_1$  and  $\Sigma_2$ .

Furthermore let  $F_1, F_2$  be a pair of conjunctions of literals over  $\mathcal{T}_1$  and  $\mathcal{T}_2$ , respectively, both compatible with some equivalence  $\sim$  on the shared variables of  $F_1$  and  $F_2$ .

Then  $F_1 \wedge F_2$  is  $(\mathcal{T}_1 \cup \mathcal{T}_2)$ -satisfiable if and only if each  $F_i$  is  $\mathcal{T}_i$ -satisfiable.

## The Nelson–Oppen Algorithm (Non-determ.)

---

Theorem 1.6:

The non-deterministic Nelson–Oppen algorithm is terminating and complete for deciding satisfiability of pure conjunctions of literals  $F_1$  and  $F_2$  over  $\mathcal{T}_1 \cup \mathcal{T}_2$  for signature-disjoint, stably infinite theories  $\mathcal{T}_1$  and  $\mathcal{T}_2$ .

Proof:

Suppose that  $F_1, F_2$  is irreducible by the inference rules of the Nelson–Oppen algorithm. Applying the amalgamation lemma in combination with Prop. 1.4 we infer that  $F_1, F_2$  is satisfiable w. r. t.  $\mathcal{T}_1 \cup \mathcal{T}_2$ .

# Convexity

---

The number of possible equivalences of shared variables grows superexponentially with the number of shared variables, so enumerating all possible equivalences non-deterministically is going to be inefficient.

A much faster variant of the Nelson–Oppen algorithm exists for convex theories.

# Convexity

---

A first-order theory  $\mathcal{T}$  is called **convex w. r. t. equations**,  
if for every conjunction  $\Gamma$  of  $\Sigma$ -equations and non-equational  $\Sigma$ -literals and  
for all  $\Sigma$ -equations  $A_i$  ( $1 \leq i \leq n$ ),  
whenever  $\mathcal{T} \models \forall \vec{x} (\Gamma \rightarrow A_1 \vee \dots \vee A_n)$ , then there exists  
some index  $j$  such that  $\mathcal{T} \models \forall \vec{x} (\Gamma \rightarrow A_j)$ .

# Convexity

---

Theorem 1.7:

If a first-order theory  $\mathcal{T}$  is convex w. r. t. equations and has no trivial models (i. e., models with only one element), then  $\mathcal{T}$  is stably infinite.

# Convexity

---

Lemma 1.8:

Suppose  $\mathcal{T}$  is convex,  $F$  a conjunction of literals, and  $S$  a subset of its variables.

Let, for any pair of variables  $x_i$  and  $x_j$  in  $S$ ,  
 $x_i \sim x_j$  if and only if  $\mathcal{T} \models \forall \vec{x} (F \rightarrow x_i \approx x_j)$ .

Then  $F$  is compatible with  $\sim$ .

# The Nelson–Oppen Algorithm (Determin./Convex)

---

Unsat:

$$\frac{F_1, F_2}{\perp}$$

if  $\exists \vec{x} F_i$  is unsatisfiable w. r. t.  $\mathcal{T}_i$  for some  $i$ .

Propagate:

$$\frac{F_1, F_2}{F_1 \wedge (x \approx y), F_2 \wedge (x \approx y)}$$

if  $x$  and  $y$  are two different variables appearing in both  $F_1$  and  $F_2$  such that

$\mathcal{T}_1 \models \forall \vec{x} (F_1 \rightarrow x \approx y)$  and  $\mathcal{T}_2 \not\models \forall \vec{x} (F_2 \rightarrow x \approx y)$   
or  $\mathcal{T}_2 \models \forall \vec{x} (F_2 \rightarrow x \approx y)$  and  $\mathcal{T}_1 \not\models \forall \vec{x} (F_1 \rightarrow x \approx y)$ .

## The Nelson–Oppen Algorithm (Determin./Convex)

Theorem 1.9:

If  $\mathcal{T}_1$  and  $\mathcal{T}_2$  are signature-disjoint theories that are convex w. r. t. equations and have no trivial models,

then the deterministic Nelson–Oppen algorithm is terminating, sound and complete for deciding satisfiability of pure conjunctions of literals  $F_1$  and  $F_2$  over  $\mathcal{T}_1 \cup \mathcal{T}_2$ .

## The Nelson–Oppen Algorithm (Determin./Convex)

Corollary 1.10:

The deterministic Nelson–Oppen algorithm for convex theories requires at most  $O(n^3)$  calls to the individual decision procedures for the component theories, where  $n$  is the number of shared variables.

## Iterating Nelson–Oppen

---

The Nelson–Oppen combination procedures can be iterated to work with more than two component theories by virtue of the following observations where signature disjointness is assumed:

Theorem 1.11:

If  $\mathcal{T}_1$  and  $\mathcal{T}_2$  are stably infinite, then so is  $\mathcal{T}_1 \cup \mathcal{T}_2$ .

# Iterating Nelson–Oppen

---

Lemma 1.12:

A first-order theory  $\mathcal{T}$  is convex w. r. t. equations if and only if for every conjunction  $\Gamma$  of  $\Sigma$ -equations and non-equational  $\Sigma$ -literals and for all equations  $x_i \approx x'_i$  ( $1 \leq i \leq n$ ), whenever  $\mathcal{T} \models \forall \vec{x} (\Gamma \rightarrow x_1 \approx x'_1 \vee \dots \vee x_n \approx x'_n)$ , then there exists some index  $j$  such that  $\mathcal{T} \models \forall \vec{x} (\Gamma \rightarrow x_j \approx x'_j)$ .

## Iterating Nelson–Oppen

---

Lemma 1.13:

Let  $\mathcal{T}$  be a first-order theory that is convex w. r. t. equations. Let  $F$  is a conjunction of literals; let  $F^-$  be the conjunction of all negative equational literals in  $F$  and let  $F^+$  be the conjunction of all remaining literals in  $F$ . If  $\mathcal{T} \models \forall \vec{x} (F \rightarrow x \approx y)$ , then  $\exists \vec{x} F$  is  $\mathcal{T}$ -unsatisfiable or  $\mathcal{T} \models \forall \vec{x} (F^+ \rightarrow x \approx y)$ .

Theorem 1.14:

If  $\mathcal{T}_1$  and  $\mathcal{T}_2$  are convex w. r. t. equations and do not have trivial models, then so is  $\mathcal{T}_1 \cup \mathcal{T}_2$ .

# Extensions

---

Many-sorted logics:

*read/2* becomes  $read : array \times int \rightarrow data$ .

*write/3* becomes  $write : array \times int \times data \rightarrow array$ .

Variables:  $x : data$

Only one declaration per function/predicate/variable symbol.

All terms, atoms, substitutions must be well-sorted.

Algebras:

Instead of universe  $U_{\mathcal{A}}$ , one set per sort:  $array_{\mathcal{A}}$ ,  $int_{\mathcal{A}}$ .

Interpretations of function and predicate symbols correspond to their declarations:  $read_{\mathcal{A}} : array_{\mathcal{A}} \times int_{\mathcal{A}} \rightarrow data_{\mathcal{A}}$

# Extensions

---

If we consider combinations of theories with shared sorts but disjoint function and predicate symbols, then we get essentially the same combination results as before.

However, stable infiniteness and/or convexity are only required for the shared sorts.

# Extensions

---

Non-stably infinite theories:

If we impose stronger conditions on one theory, we can relax the conditions on the other one.

For instance, EUF can be combined with any other theory; stable infiniteness is not required.

E.g.: Strongly polite theories, shiny theories, flexible theories.

## Strong Politeness

---

A theory  $\mathcal{T}$  is called **smooth**, if every quantifier-free formula that has a  $\mathcal{T}$ -model with some (finite or infinite) cardinality  $\kappa_0$  has also  $\mathcal{T}$ -models with cardinality  $\kappa$  for every  $\kappa \geq \kappa_0$ .

# Strong Politeness

---

A theory  $\mathcal{T}$  is called **finitely witnessable**, if there is a computable function  $wit$  that maps every quantifier-free formula  $F$  to a quantifier-free formula  $G$  such that

1.  $F$  and  $\exists \vec{w} G$  are  $\mathcal{T}$ -equivalent, where  $\vec{w} = \text{var}(G) \setminus \text{var}(F)$ ,
2. if  $G \wedge \Delta$  is  $\mathcal{T}$ -satisfiable for some arrangement  $\Delta$ , then there is a  $\mathcal{T}$ -model  $\mathcal{A}$  and an assignment  $\beta$  such that  $\mathcal{A}, \beta \models (G \wedge \Delta)$  and  $U_{\mathcal{A}} = \{ \beta(x) \mid x \in \text{var}(G \wedge \Delta) \}$

# Strong Politeness

---

A theory  $\mathcal{T}$  is called **strongly polite**, if it is smooth and finitely witnessable.

Theorem 1.15 (Barrett & Jovanović):

We can combine two theories  $\mathcal{T}_1$  and  $\mathcal{T}_2$  if one of them is strongly polite.

Again, in the many-sorted case, smoothness and finite witnessability must hold for all the shared sorts.

# Strong Politeness

---

Non-disjoint combinations:

Have to ensure that both decision procedures interpret shared symbols in a compatible way.

Some results, e. g. by Ghilardi, using strong model theoretical conditions on the theories.

# Another Combination Method

---

Shostak's method:

Applicable to combinations of EUF and *solvable* theories.

## Another Combination Method

---

A  $\Sigma$ -theory  $\mathcal{T}$  is called **solvable**, if there exists an effectively computable function *solve* such that, for any  $\mathcal{T}$ -equation  $s \approx t$ :

(A)  $\text{solve}(s \approx t) = \perp$  if and only if  $\mathcal{T} \models \forall \vec{x} (s \not\approx t)$ ;

(B)  $\text{solve}(s \approx t) = \emptyset$  if and only if  $\mathcal{T} \models \forall \vec{x} (s \approx t)$ ;  
and otherwise

(C)  $\text{solve}(s \approx t) = \{x_1 \approx u_1, \dots, x_n \approx u_n\}$ , where

– the  $x_i$  are pairwise different variables occurring in  $s \approx t$ ;

– the  $x_i$  do not occur in the  $u_j$ ; and

–  $\mathcal{T} \models \forall \vec{x} ((s \approx t) \leftrightarrow \exists \vec{y} (x_1 \approx u_1 \wedge \dots \wedge x_n \approx u_n))$ , where  $\vec{y}$  are the variables occurring in one of the  $u_j$  but not in  $s \approx t$ , and  $\vec{x} \cap \vec{y} = \emptyset$ .

## Another Combination Method

---

Additionally useful (but not required):

A canonizer, that is, a function that simplifies terms by computing some unique normal form

# Another Combination Method

---

Main idea of the procedure:

If  $s \approx t$  is a positive equation and

$$\text{solve}(s \approx t) = \{x_1 \approx u_1, \dots, x_n \approx u_n\},$$

replace  $s \approx t$  by  $x_1 \approx u_1 \wedge \dots \wedge x_n \approx u_n$

and use these equations to eliminate the  $x_i$  elsewhere.

Practical problem:

Solvability is a rather restrictive condition.

## Part 2: Satisfiability Modulo Theories (SMT)

---

So far:

decision procedures for satisfiability for various fragments of first-order theories;

often only for ground conjunctions of literals.

Goals:

extend decision procedures efficiently to ground CNF formulas;

later: extend to non-ground formulas

(we will often lose completeness, however).

## 2.1 The CDCL( $\mathcal{T}$ ) Procedure

---

Goal:

Given a propositional formula in CNF (or alternatively, a finite set  $N$  of clauses), where the atoms represent ground formulas over some theory  $\mathcal{T}$ , check whether it is satisfiable in  $\mathcal{T}$  (and optionally: output *one* solution, if it is satisfiable).

Assumption:

As in the propositional case, clauses contain neither duplicated literals nor complementary literals.

## The CDCL(T) Procedure

---

For propositional CDCL (“Conflict-Driven Clause Learning”), we have considered partial valuations, i. e., partial mappings from propositional variables to truth values.

A partial valuation  $\mathcal{A}$  corresponds to a set  $M$  of literals that does not contain complementary literals, and vice versa:

$\mathcal{A}(L)$  is true, if  $L \in M$ .

$\mathcal{A}(L)$  is false, if  $\bar{L} \in M$ .

$\mathcal{A}(L)$  is undefined, if neither  $L \in M$  nor  $\bar{L} \in M$ .

## The CDCL( $\mathcal{T}$ ) Procedure

---

We will now consider partial mappings from ground  $\mathcal{T}$ -atoms to truth values (which correspond to sets of  $\mathcal{T}$ -literals).

In order to check whether a (partial) valuation is permissible, we identify the valuation  $\mathcal{A}$  or the set  $M$  with the conjunction of all literals in  $M$ :

The valuation  $\mathcal{A}$  or the set  $M$  is called  $\mathcal{T}$ -satisfiable, if the literals in  $M$  have a  $\mathcal{T}$ -model.

## The CDCL( $\mathcal{T}$ ) Procedure

---

Since the elements of  $M$  can be interpreted both as propositional variables and as ground  $\mathcal{T}$ -formulas, we have to distinguish between two notions of entailment:

We write  $M \models F$  if  $F$  is entailed by  $M$  propositionally.

We write  $M \models_{\mathcal{T}} F$  if the ground  $\mathcal{T}$ -formulas represented by  $M$  entail  $F$ .

$M$  is called a  $\mathcal{T}$ -model of  $F$ , if it is  $\mathcal{T}$ -satisfiable and  $M \models_{\mathcal{T}} F$ .

We write  $F \models_{\mathcal{T}} G$ , if the formula  $F$  entails  $G$  w. r. t.  $\mathcal{T}$ , that is, if every  $\mathcal{T}$ -model of  $F$  is also a model of  $G$ .

# Idea

---

Naive Approach:

Use CDCL to find a propositionally satisfying valuation.

If the valuation found is  $\mathcal{T}$ -satisfiable, stop;  
otherwise continue CDCL search.

Note: The CDCL procedure may *not* use “pure literal” checks.

# Idea

---

Improvements:

Check already partial valuations for  $\mathcal{T}$ -satisfiability.

If  $\mathcal{T}$ -decision procedure yields explanations,  
use them for non-chronological backjumping.

If  $\mathcal{T}$ -decision procedure can provide  $\mathcal{T}$ -entailed literals,  
use them for propagation.

Since  $\mathcal{T}$ -satisfiability checks may be costly,  
learn clauses that incorporate useful  $\mathcal{T}$ -knowledge,  
in particular explanations for backjumping.

# CDCL( $\mathcal{T}$ )

---

The “CDCL Modulo Theories” procedure is modelled by a transition relation  $\Rightarrow_{\text{CDCL}(\mathcal{T})}$  on a set of states.

States:

- *fail*
- $M \parallel N$ ,

where  $M$  is a *list of annotated literals* (“*trail*”) and  $N$  is a set of clauses.

Annotated literal:

- $L$ : deduced literal, due to propagation.
- $L^d$ : decision literal (guessed literal).

# CDCL(T) Rules from CDCL

---

Unit Propagate:

$$M \parallel N \cup \{C \vee L\} \Rightarrow_{\text{CDCL}(\mathcal{T})} M L \parallel N \cup \{C \vee L\}$$

if  $C$  is false under  $M$  and  $L$  is undefined under  $M$ .

Decide:

$$M \parallel N \Rightarrow_{\text{CDCL}(\mathcal{T})} M L^d \parallel N$$

if  $L$  is undefined under  $M$ .

Fail:

$$M \parallel N \cup \{C\} \Rightarrow_{\text{CDCL}(\mathcal{T})} \text{fail}$$

if  $C$  is false under  $M$  and  $M$  contains no decision literals.

## Specific CDCL( $\mathcal{T}$ ) Rules

---

$\mathcal{T}$ -Learn:

$$M \parallel N \Rightarrow_{\text{CDCL}(\mathcal{T})} M \parallel N \cup \{C\}$$

if  $N \models_{\mathcal{T}} C$  and each atom of  $C$  occurs in  $N$  or  $M$ .

$\mathcal{T}$ -Forget:

$$M \parallel N \cup \{C\} \Rightarrow_{\text{CDCL}(\mathcal{T})} M \parallel N$$

if  $N \models_{\mathcal{T}} C$ .

$\mathcal{T}$ -Propagate:

$$M \parallel N \Rightarrow_{\text{CDCL}(\mathcal{T})} M L \parallel N$$

if  $M \models_{\mathcal{T}} L$  where  $L$  is undefined in  $M$ , and  $L$  or  $\bar{L}$  occurs in  $N$ .

## Specific CDCL( $\mathcal{T}$ ) Rules

---

$\mathcal{T}$ -Backjump:

$$M' L^d M'' \parallel N \Rightarrow_{\text{CDCL}(\mathcal{T})} M' L' \parallel N$$

if  $M' L^d M'' \models \neg C$  for some  $C \in N$

and if there is some “backjump clause”  $C' \vee L'$  such that

$N \models_{\mathcal{T}} C' \vee L'$  and  $M' \models \neg C'$ ,

$L'$  is undefined under  $M'$ , and

$L'$  or  $\overline{L'}$  occurs in  $N$  or in  $M' L^d M''$ .

Note: We don't need a special rule to handle the case that  $M' L^d M'' \models_{\mathcal{T}} \perp$ .

If the trail contains a  $\mathcal{T}$ -inconsistent subset, we can always add the negation of that subset using  $\mathcal{T}$ -Learn and apply  $\mathcal{T}$ -Backjump afterwards.

## CDCL( $\mathcal{T}$ ) Properties

---

The system  $\text{CDCL}(\mathcal{T})$  consists of the rules Decide, Fail, Unit Propagate,  $\mathcal{T}$ -Propagate,  $\mathcal{T}$ -Backjump,  $\mathcal{T}$ -Learn and  $\mathcal{T}$ -Forget.

Lemma 2.1:

If we reach a state  $M \parallel N$  starting from  $\emptyset \parallel N$ , then:

- (1)  $M$  does not contain complementary literals.
- (2) Every deduced literal  $L$  in  $M$  follows from  $\mathcal{T}$ ,  $N$ , and decision literals occurring before  $L$  in  $M$ .

# CDCL(T) Properties

---

Lemma 2.2:

If no clause is learned infinitely often, then every derivation starting from  $\emptyset \parallel N$  terminates.

# CDCL( $\mathcal{T}$ ) Properties

---

Lemma 2.3:

If  $\emptyset \parallel N \Rightarrow_{\text{CDCL}(\mathcal{T})}^* M \parallel N'$  and there is some conflicting clause in  $M \parallel N'$ , that is,  $M \models \neg C$  for some clause  $C$  in  $N'$ , then either Fail or  $\mathcal{T}$ -Backjump applies to  $M \parallel N'$ .

# CDCL( $\mathcal{T}$ ) Properties

---

Lemma 2.4:

If  $\emptyset \parallel N \Rightarrow_{\text{CDCL}(\mathcal{T})}^* M \parallel N'$  and  $M$  is  $\mathcal{T}$ -unsatisfiable, then either there is a conflicting clause in  $M \parallel N'$ , or else  $\mathcal{T}$ -Learn applies to  $M \parallel N'$ , generating a conflicting clause.

## CDCL( $\mathcal{T}$ ) Properties

---

Theorem 2.5:

Consider a derivation  $\emptyset \parallel N \Rightarrow_{\text{CDCL}(\mathcal{T})}^* S$ , where no more rules of the CDCL( $\mathcal{T}$ ) procedure are applicable to  $S$  except  $\mathcal{T}$ -Learn or  $\mathcal{T}$ -Forget, and if  $S$  has the form  $M \parallel N'$  then  $M$  is  $\mathcal{T}$ -satisfiable. Then

- (1) If  $S$  has the form  $M \parallel N'$ , then  $M$  is a  $\mathcal{T}$ -model of  $N$  and  $N'$ .
- (2) If  $S$  is *fail* then  $N$  and  $N'$  are  $\mathcal{T}$ -unsatisfiable.

## The Solver Interface

---

The general  $\text{CDCL}(\mathcal{T})$  procedure has to be connected to a “Solver” for  $\mathcal{T}$ , a theory module that performs *at least*  $\mathcal{T}$ -satisfiability checks.

The solver is initialized with a list of all literals occurring in the input of the  $\text{CDCL}(\mathcal{T})$  procedure.

Internally, it keeps a stack  $I$  of theory literals that is initially empty. The solver performs the following operations on  $I$ :

# The Solver Interface

---

SetTrue( $L$ :  $\mathcal{T}$ -Literal):

Check whether  $I \cup \{L\}$  is  $\mathcal{T}$ -satisfiable.

If no: return an explanation for  $\bar{L}$ , that is, a subset  $J$  of  $I$  such that  $J \models_{\mathcal{T}} \bar{L}$ .

If yes: push  $L$  on  $I$ .

Optionally: Return a list of literals that are  $\mathcal{T}$ -consequences of  $I \cup \{L\}$  (and have not yet been detected before).

Note: Depending on  $\mathcal{T}$ , detecting (all)  $\mathcal{T}$ -consequences may be very cheap or very expensive.

# The Solver Interface

---

Backtrack( $n: \mathbb{N}$ ):

Pop  $n$  literals from  $I$ .

# The Solver Interface

---

Explanation( $L$ :  $\mathcal{T}$ -Literal):

Return an explanation for  $L$ , that is, a subset  $J$  of  $I$  such that  $J \models_{\mathcal{T}} L$ .

We assume that  $L$  has been returned previously as a result of some SetTrue( $L'$ ) operation.

No literal of  $J$  may occur in  $I$  after  $L'$ .

# Computing Backjump Clauses

---

Backjump clauses for a conflict can then be computed as in the propositional case:

Start with the conflicting clause.

Resolve with the clauses used for Unit Propagate or the explanations produced by the solver until a backjump clause (or  $\perp$ ) is found.

## 2.2 Heuristic Instantiation

---

CDCL( $\mathcal{T}$ ) is limited to ground (or existentially quantified) formulas. Even if we have decidability for more than the ground fragment of a theory  $\mathcal{T}$ , we cannot use this in CDCL( $\mathcal{T}$ ).

Most current SMT implementations offer a limited support for universally quantified formulas by heuristic instantiation.

# Heuristic Instantiation

---

Goal:

Create potentially useful ground instances of universally quantified clauses and add them to the given ground clauses.

Idea (Detlefs, Nelson, Saxe: Simplify):

Select subset of the terms (or atoms) in  $\forall \vec{x} C$  as “trigger” (automatically, but can be overridden manually).

If there is a ground instance  $C\theta$  of  $\forall \vec{x} C$  such that  $t\theta$  occurs (modulo congruence) in the current set of ground clauses for every  $t \in \text{trigger}(C)$ , add  $C\theta$  to the set of ground clauses (incrementally).

# Heuristic Instantiation

---

Conditions for trigger terms (or atoms):

- (1) Every quantified variable of the clause occurs in some trigger term (therefore more than one trigger term may be necessary).
- (2) A trigger term is not a variable itself.
- (3) A trigger is not explicitly forbidden by the user.
- (4) There is no larger instance of the term in the formula:  
(If  $f(x)$  were selected as a trigger in  $\forall x P(f(x), f(g(x)))$ , a ground term  $f(a)$  would produce an instance  $P(f(a), f(g(a)))$ , which would produce  $P(f(g(a)), f(g(g(a))))$ , and so on.)
- (5) No proper subterm satisfies (1)–(4).

## Heuristic Instantiation

---

Also possible (but expensive, therefore only in restricted form): Theory matching

The ground atom  $P(a)$  is not an instance of the trigger atom  $P(x + 1)$ ; it is however equivalent (in linear algebra) to  $P((a - 1) + 1)$ , which *is* an instance and may therefore produce a new ground clause.

# Heuristic Instantiation

---

Heuristic instantiation is obviously incomplete

e. g., it does not find the contradiction for  $f(x, a) \approx x$ ,  $f(b, y) \approx y$ ,  
 $a \neq b$

but it is quite useful in practice:

modern implementations: CVC, Yices, Z3.

## 2.3 Local Theory Extensions

---

Under certain circumstances, instantiating universally quantified variables with “known” ground terms is sufficient for completeness.

Scenario:

$\Sigma_0 = (\Omega_0, \Pi_0)$ : base signature;

$\mathcal{T}_0$ :  $\Sigma_0$ -theory.

$\Sigma_1 = (\Omega_0 \cup \Omega_1, \Pi_0)$ : signature extension;

$K$ : universally quantified  $\Sigma_1$ -clauses;

$G$ : ground clauses.

# Local Theory Extensions

---

Assumption: clauses in  $G$  are  $\Sigma_1$ -flat and  $\Sigma_1$ -linear:

only constants as arguments of  $\Omega_1$ -symbols,

if a constant occurs in two terms below an  $\Omega_1$ -symbol, then the two terms are identical,

no term contains the same constant twice below an  $\Omega_1$ -symbol.

# Local Theory Extensions

---

Example: Monotonic functions over  $\mathbb{Z}$ .

$\mathcal{T}_0$ : Linear integer arithmetic.

$$\Omega_1 = \{f/1\}.$$

$$K = \{ \forall x, y (\neg x \leq y \vee f(x) \leq f(y)) \}.$$

$$G = \{ f(3) \geq 6, f(5) \leq 9 \}.$$

Observation: If we choose interpretations for  $f(3)$  and  $f(5)$  that satisfy the  $G$  and monotonicity axiom, then it is always possible to define  $f$  for all remaining integers such that the monotonicity axiom is satisfied.

# Local Theory Extensions

---

Example: Strictly monotonic functions over  $\mathbb{Z}$ .

$\mathcal{T}_0$ : Linear integer arithmetic.

$$\Omega_1 = \{f/1\}.$$

$$K = \{ \forall x, y (\neg x < y \vee f(x) < f(y)) \}.$$

$$G = \{ f(3) > 6, f(5) < 9 \}.$$

Observation: Even though we can choose interpretations for  $f(3)$  and  $f(5)$  that satisfy  $G$  and the strict monotonicity axiom (map  $f(3)$  to 7 and  $f(5)$  to 8), we cannot define  $f(4)$  such that the strict monotonicity axiom is satisfied.

# Local Theory Extensions

---

To formalize the idea, we need partial algebras:

like (usual) total algebras, but  $f_{\mathcal{A}}$  may be a partial function.

## Local Theory Extensions

---

There are several ways to define equality in partial algebras (strong equality, Evans equality, weak equality, etc.). Here we use weak equality:

an equation  $s \approx t$  holds w. r. t.  $\mathcal{A}$  and  $\beta$  if both  $\mathcal{A}(\beta)(s)$  and  $\mathcal{A}(\beta)(t)$  are defined and equal or if at least one of them is undefined;

a negated equation  $s \not\approx t$  holds w. r. t.  $\mathcal{A}$  and  $\beta$  if both  $\mathcal{A}(\beta)(s)$  and  $\mathcal{A}(\beta)(t)$  are defined and different or if at least one of them is undefined.

If a partial algebra  $\mathcal{A}$  satisfies a set of formulas  $N$  w. r. t. weak equality, it is called a weak partial model of  $N$ .

## Local Theory Extensions

---

A partial algebra  $\mathcal{A}$  embeds weakly into a partial algebra  $\mathcal{B}$  if there is an injective total mapping  $h : U_{\mathcal{A}} \rightarrow U_{\mathcal{B}}$  such that if  $f_{\mathcal{A}}(a_1, \dots, a_n)$  is defined in  $\mathcal{A}$  then  $f_{\mathcal{B}}(h(a_1), \dots, h(a_n))$  is defined in  $\mathcal{B}$  and equal to  $h(f_{\mathcal{A}}(a_1, \dots, a_n))$ .

## Local Theory Extensions

---

A theory extension  $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup K$  is called *local*, if for every set  $G$ ,  $\mathcal{T}_0 \cup K \cup G$  is satisfiable if and only if  $\mathcal{T}_0 \cup K[G] \cup G$  has a (partial) model, where  $K[G]$  is the set of instances of clauses in  $K$  in which all terms starting with an  $\Omega_1$ -symbol are ground terms occurring in  $K$  or  $G$ .

If every weak partial model of  $\mathcal{T}_0 \cup K$  can be embedded into a total model, then the theory extension  $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup K$  is local (Sofronie-Stokkermans 2005).

Note: There are many variants of partial models and embeddings corresponding to different kinds of locality.

# Local Theory Extensions

---

Examples of local theory extensions:

free functions,

constructors/selectors,

monotonic functions,

Lipschitz functions.

## 2.4 Goal-driven Instantiation

---

Instantiation is used to refute the current model discovered by the ground solver.

Rather than a fast but loosely guided instantiation technique, we can search for the most suitable instance if it exists.

# Goal-driven Instantiation

---

Scenario:

$M$ : a model of the ground formula returned by the ground SMT solver.

$Q$ : the set of universally quantified clauses contained in the original input.

Problem:

Find a clause  $\forall x C \in Q$  and a grounding substitution  $\sigma$  such that  $M \cup C\sigma$  is unsatisfiable, if it exists.

# E-ground (Dis)unification Problem

---

Given

$E$ : a set of ground equality literals,

$N$ : a set of equality literals,

find  $\sigma$  such that  $E \models N\sigma$ .

## E-ground (Dis)unification Problem

The E-ground (dis)unification problem can be used to encode the goal-driven instantiation problem:

For  $M$  and each  $\forall x C \in Q$ , try to solve the E-ground (dis)unification problem  $M \models (\neg C)\sigma$ .

# Congruence Closure with Free Variables

---

CCFV (Barbosa et al, 2017) decomposes  $N$  into sets of smaller constraints by replacing terms with equivalent smaller ones until either

1. a variable assignment is possible, and the decomposition restarts afterwards,
2. a contradiction occurs, and the corresponding search branch is closed,
3. a substitution satisfying the problem is found.

# Congruence Closure with Free Variables

---

CCFV is sound, complete and terminating for the E-ground (dis)unification problem.

Modern implementations: CVC4, VeriT.

## Part 3: Superposition

---

First-order calculi considered so far:

Resolution: for first-order clauses without equality.

(Unfailing) Knuth-Bendix Completion: for unit equations.

Goal:

Combine the ideas of ordered resolution (overlap maximal literals in a clause) and Knuth-Bendix completion (overlap maximal sides of equations) to get a calculus for equational clauses.

## 3.1 Recapitulation

---

First-order logic:

Atom: either  $P(s_1, \dots, s_m)$  with  $P \in \Pi$  or  $s \approx t$ .

Literal: Atom or negated atom.

Clause: (possibly empty) disjunction of literals  
(all variables implicitly universally quantified).

# Recapitulation

---

Refutational theorem proving:

For refutational theorem proving, it suffices to consider sets of clauses: every first-order formula  $F$  can be translated into a set of clauses  $N$  such that  $F$  is unsatisfiable if and only if  $N$  is unsatisfiable.

In the non-equational case, unsatisfiability can for instance be checked using the (ordered) resolution calculus.

# Recapitulation

---

(Ordered) resolution: inference rules:

Ground case:

$$\text{Resolution: } \frac{D' \vee A \quad C' \vee \neg A}{D' \vee C'}$$

Non-ground case:

$$\frac{D' \vee A \quad C' \vee \neg A'}{(D' \vee C')\sigma}$$

where  $\sigma = \text{mgu}(A, A')$ .

$$\text{Factoring: } \frac{C' \vee A \vee A}{C' \vee A}$$

$$\frac{C' \vee A \vee A'}{(C' \vee A)\sigma}$$

where  $\sigma = \text{mgu}(A, A')$ .

# Recapitulation

---

Ordering restrictions:

Let  $\succ$  be a well-founded and total ordering on ground atoms.

Literal ordering  $\succ_L$ :

compares literals by comparing lexicographically first the respective atoms using  $\succ$  and then their polarities (negative  $>$  positive).

Clause ordering  $\succ_C$ :

compares clauses by comparing their multisets of literals using the multiset extension of  $\succ_L$ .

# Recapitulation

---

Ordering restrictions (ground case):

Inference are necessary only if the following conditions are satisfied:

- The left premise of a Resolution inference is not larger than or equal to the right premise.
- The literals that are involved in the inferences ( $[\neg] A$ ) are maximal in the respective clauses  
(strictly maximal for the left premise of Resolution).

# Recapitulation

---

Ordering restrictions (non-ground case):

Define the atom ordering  $\succ$  also for non-ground atoms.

Need stability under substitutions:  $A \succ B$  implies  $A\sigma \succ B\sigma$ .

Note:  $\succ$  cannot be total on non-ground atoms.

For literals involved in inferences we have the same maximality requirements as in the ground case.

# Recapitulation

---

Resolution is (even with ordering restrictions) refutationally complete:

Dynamic view of refutational completeness:

If  $N$  is unsatisfiable ( $N \models \perp$ ) then *fair* derivations from  $N$  produce  $\perp$ .

Static view of refutational completeness:

If  $N$  is *saturated*, then  $N$  is unsatisfiable if and only if  $\perp \in N$ .

# Recapitulation

---

Proving refutational completeness for the ground case:

We have to show:

If  $N$  is saturated (i. e., if sufficiently many inferences have been computed), and  $\perp \notin N$ , then  $N$  has a model.

# Recapitulation

---

Constructing a candidate interpretation:

Suppose that  $N$  be saturated and  $\perp \notin N$ .

We inspect all clauses in  $N$  in ascending order and construct a sequence of Herbrand interpretations

(starting with the empty interpretation: all atoms are false).

If a clause  $C$  is false in the current interpretation, and has a positive and strictly maximal literal  $A$ , then extend the current interpretation such that  $C$  becomes true: add  $A$  to the current interpretation.

(Then  $C$  is called *productive*.)

Otherwise, leave the current interpretation unchanged.

# Recapitulation

---

The sequence of interpretations has the following properties:

- (1) If an atom is true in some interpretation, then it remains true in all future interpretations.
- (2) If a clause is true at the time where it is inspected, then it remains true in all future interpretations.
- (3) If a clause  $C = C' \vee A$  is productive, then  $C$  remains true and  $C'$  remains false in all future interpretations.

Show by induction: if  $N$  is saturated and  $\perp \notin N$ , then every clause in  $N$  is either true at the time where it is inspected or productive.

# Recapitulation

---

Note:

For the induction proof, it is not necessary that the conclusion of an inference is contained in  $N$ .

It is sufficient that it is redundant w. r. t.  $N$ .

$N$  is called *saturated up to redundancy* if the conclusion of every inference from clauses in  $N \setminus Red(N)$  is contained in  $N \cup Red(N)$ .

# Recapitulation

---

Proving refutational completeness for the non-ground case:

If  $C_i\theta$  is a ground instance of the clause  $C_i$  for  $i \in \{0, \dots, n\}$  and

$$\frac{C_n, \dots, C_1}{C_0}$$

and

$$\frac{C_n\theta, \dots, C_1\theta}{C_0\theta}$$

are inferences, then the latter inference is called a **ground instance** of the former.

# Recapitulation

---

For a set  $N$  of clauses, let  $G_{\Sigma}(N)$  be the set of all ground instances of clauses in  $N$ .

Construct the interpretation from the set  $G_{\Sigma}(N)$  of all ground instances of clauses in  $N$ :

$N$  is saturated and does not contain  $\perp$

$\Rightarrow G_{\Sigma}(N)$  is saturated and does not contain  $\perp$

$\Rightarrow G_{\Sigma}(N)$  has a Herbrand model  $I$

$\Rightarrow I$  is a model of  $N$ .

# Recapitulation

---

It is possible to encode an arbitrary predicate  $P$  using a function  $f_P$  and a new constant  $true$ :

$$\begin{aligned} P(t_1, \dots, t_n) &\quad \rightsquigarrow \quad f_P(t_1, \dots, t_n) \approx true \\ \neg P(t_1, \dots, t_n) &\quad \rightsquigarrow \quad \neg f_P(t_1, \dots, t_n) \approx true \end{aligned}$$

In equational logic it is therefore sufficient to consider the case that  $\Pi = \emptyset$ , i. e., equality is the only predicate symbol.

Abbreviation:  $s \not\approx t$  instead of  $\neg s \approx t$ .

## 3.2 The Superposition Calculus – Informally

---

Conventions:

From now on:  $\Pi = \emptyset$  (equality is the only predicate).

Inference rules are to be read modulo symmetry of the equality symbol.

We will first explain the ideas and motivations behind the superposition calculus and its completeness proof. Precise definitions will be given later.

# The Superposition Calculus – Informally

---

Ground inference rules:

Pos. Superposition: 
$$\frac{D' \vee t \approx t' \quad C' \vee s[t] \approx s'}{D' \vee C' \vee s[t'] \approx s'}$$

Neg. Superposition: 
$$\frac{D' \vee t \approx t' \quad C' \vee s[t] \not\approx s'}{D' \vee C' \vee s[t'] \not\approx s'}$$

Equality Resolution: 
$$\frac{C' \vee s \not\approx s}{C'}$$

(Note: We will need one further inference rule.)

# The Superposition Calculus – Informally

---

Ordering wishlist:

Like in resolution, we want to perform only inferences between (strictly) maximal literals.

Like in completion, we want to perform only inferences between (strictly) maximal sides of literals.

Like in resolution, in inferences with two premises, the left premise should not be larger than the right one.

Like in resolution and completion, the conclusion should then be smaller than the larger premise.

The ordering should be total on ground literals.

# The Superposition Calculus – Informally

---

Consequences:

The literal ordering must depend primarily on the larger term of an equation.

As in the resolution case, negative literals must be a bit larger than the corresponding positive literals.

Additionally, we need the following property:

If  $s \succ t \succ u$ , then  $s \not\approx u$  must be larger than  $s \approx t$ .

In other words, we must compare first the larger term, then the polarity, and finally the smaller term.

# The Superposition Calculus – Informally

---

The following construction has the required properties:

Let  $\succ$  be a *reduction ordering that is total on ground terms*.

To a positive literal  $s \approx t$ , we assign the multiset  $\{s, t\}$ ,  
to a negative literal  $s \not\approx t$  the multiset  $\{s, s, t, t\}$ .

The **literal ordering**  $\succ_L$  compares these multisets using the multiset extension of  $\succ$ .

The **clause ordering**  $\succ_C$  compares clauses by comparing their multisets of literals using the multiset extension of  $\succ_L$ .

# The Superposition Calculus – Informally

---

Constructing a candidate interpretation:

We want to use roughly the same ideas as in the completeness proof for resolution.

But: a Herbrand interpretation does not work for equality:

The equality symbol  $\approx$  must be interpreted by equality in the interpretation.

# The Superposition Calculus – Informally

---

Solution: Productive clauses contribute ground rewrite rules to a TRS  $R$ .

The interpretation has the universe  $T_{\Sigma}(\emptyset)/R = T_{\Sigma}(\emptyset)/\approx_R$ ;

a ground atom  $s \approx t$  holds in the interpretation if and only if  $s \approx_R t$   
if and only if  $s \leftrightarrow_R^* t$ .

We will construct  $R$  in such a way that it is terminating and confluent.

In this case,  $s \approx_R t$  if and only if  $s \downarrow_R t$ .

# The Superposition Calculus – Informally

---

One problem:

The completeness proof for the resolution calculus depends on the following property:

If  $C = C' \vee A$  with a strictly maximal and positive literal  $A$  is false in the current interpretation, then adding  $A$  to the current interpretation cannot make any literal of  $C'$  true.

This property does not hold for superposition:

Let  $b \succ c \succ d$ .

Assume that the current rewrite system (representing the current interpretation) contains the rule  $c \rightarrow d$ .

Now consider the clause  $b \approx d \vee b \approx c$ .

## The Superposition Calculus – Informally

---

We need a further inference rule to deal with clauses of this kind, either the “Merging Paramodulation” rule of Bachmair and Ganzinger or the following “Equality Factoring” rule due to Nieuwenhuis:

Equality Factoring: 
$$\frac{C' \vee s \approx t' \vee s \approx t}{C' \vee t \not\approx t' \vee s \approx t'}$$

Note: This inference rule subsumes the usual factoring rule.

# The Superposition Calculus – Informally

---

How do the non-ground versions of the inference rules for superposition look like?

Main idea as in the resolution calculus:

Replace identity by unifiability.

Apply the mgu to the resulting clause.

In the ordering restrictions, use  $\not\prec$  instead of  $\succ$ .

# The Superposition Calculus – Informally

---

However:

As in Knuth-Bendix completion, we do not want to consider overlaps at or below a variable position.

Consequence: there are inferences between ground instances  $D\theta$  and  $C\theta$  of clauses  $D$  and  $C$  which are *not* ground instances of inferences between  $D$  and  $C$ .

Such inferences have to be treated in a special way in the completeness proof.

## 3.3 The Superposition Calculus – Formally

---

Until now, we have seen most of the ideas behind the superposition calculus and its completeness proof.

We will now start again from the beginning giving precise definitions and proofs.

# The Superposition Calculus – Formally

---

Inference rules (part 1):

Pos. Superposition:

$$\frac{D' \vee t \approx t' \quad C' \vee s[u] \approx s'}{(D' \vee C' \vee s[t'] \approx s')\sigma}$$

where  $\sigma = \text{mgu}(t, u)$  and  $u$  is not a variable.

Neg. Superposition:

$$\frac{D' \vee t \approx t' \quad C' \vee s[u] \not\approx s'}{(D' \vee C' \vee s[t'] \not\approx s')\sigma}$$

where  $\sigma = \text{mgu}(t, u)$  and  $u$  is not a variable.

# The Superposition Calculus – Formally

---

Inference rules (part 2):

Equality Resolution: 
$$\frac{C' \vee s \not\approx s'}{C'\sigma}$$
where  $\sigma = \text{mgu}(s, s')$ .

Equality Factoring: 
$$\frac{C' \vee s' \approx t' \vee s \approx t}{(C' \vee t \not\approx t' \vee s \approx t')\sigma}$$
where  $\sigma = \text{mgu}(s, s')$ .

# The Superposition Calculus – Formally

---

Theorem 3.1:

All inference rules of the superposition calculus are **correct**, i. e., for every rule

$$\frac{C_n, \dots, C_1}{C_0}$$

we have  $\{C_1, \dots, C_n\} \models C_0$ .

Proof:

Exercise. □

# The Superposition Calculus – Formally

---

Orderings:

Let  $\succ$  be a *reduction ordering that is total on ground terms*.

To a positive literal  $s \approx t$ , we assign the multiset  $\{s, t\}$ ,  
to a negative literal  $s \not\approx t$  the multiset  $\{s, s, t, t\}$ .

The **literal ordering**  $\succ_L$  compares these multisets using the multiset extension of  $\succ$ .

The **clause ordering**  $\succ_C$  compares clauses by comparing their multisets of literals using the multiset extension of  $\succ_L$ .

# The Superposition Calculus – Formally

---

Inferences have to be computed only if the following ordering restrictions are satisfied (after applying the unifier to the premises):

- In superposition inferences, the left premise is not greater than or equal to the right one.
- The last literal in each premise is maximal in the respective premise, i. e., there exists no greater literal (strictly maximal for positive literals in superposition inferences, i. e., there exists no greater or equal literal).
- In these literals, the lhs is neither smaller nor equal than the rhs (except in equality resolution inferences).

# The Superposition Calculus – Formally

---

A ground clause  $C$  is called **redundant w. r. t. a set of ground clauses  $N$** , if it follows from clauses in  $N$  that are smaller than  $C$ .

A clause is **redundant w. r. t. a set of clauses  $N$** , if all its ground instances are redundant w. r. t.  $G_{\Sigma}(N)$ .

The set of all clauses that are redundant w. r. t.  $N$  is denoted by  **$Red(N)$** .

$N$  is called **saturated up to redundancy**, if the conclusion of every inference from clauses in  $N \setminus Red(N)$  is contained in  $N \cup Red(N)$ .

## 3.4 Superposition: Refutational Completeness

---

For a set  $E$  of ground equations,  $T_{\Sigma}(\emptyset)/E$  is an  $E$ -interpretation (or  $E$ -algebra) with universe  $\{ [t] \mid t \in T_{\Sigma}(\emptyset) \}$ .

One can show (similar to the proof of Birkhoff's Theorem) that for every *ground* equation  $s \approx t$  we have  $T_{\Sigma}(\emptyset)/E \models s \approx t$  if and only if  $s \leftrightarrow_E^* t$ .

In particular, if  $E$  is a convergent set of rewrite rules  $R$  and  $s \approx t$  is a ground equation, then  $T_{\Sigma}(\emptyset)/R \models s \approx t$  if and only if  $s \downarrow_R t$ .

By abuse of terminology, we say that an equation or clause is valid (or true) in  $R$  if and only if it is true in  $T_{\Sigma}(\emptyset)/R$ .

# Superposition: Refutational Completeness

---

## Construction of candidate interpretations

(Bachmair & Ganzinger 1990):

Let  $N$  be a set of clauses not containing  $\perp$ .

Using induction on the clause ordering we define sets of rewrite rules

$E_C$  and  $R_C$  for all  $C \in G_\Sigma(N)$  as follows:

Assume that  $E_D$  has already been defined for all  $D \in G_\Sigma(N)$  with  $D \prec_C C$ .

Then  $R_C = \bigcup_{D \prec_C C} E_D$ .

# Superposition: Refutational Completeness

---

The set  $E_C$  contains the rewrite rule  $s \rightarrow t$ , if

- (a)  $C = C' \vee s \approx t$ .
- (b)  $s \approx t$  is strictly maximal in  $C$ .
- (c)  $s \succ t$ .
- (d)  $C$  is false in  $R_C$ .
- (e)  $C'$  is false in  $R_C \cup \{s \rightarrow t\}$ .
- (f)  $s$  is irreducible w. r. t.  $R_C$ .

In this case,  $C$  is called **productive**. Otherwise  $E_C = \emptyset$ .

Finally,  $R_\infty = \bigcup_{D \in G_\Sigma(N)} E_D$ .

# Superposition: Refutational Completeness

---

Lemma 3.2:

If  $E_C = \{s \rightarrow t\}$  and  $E_D = \{u \rightarrow v\}$ , then  $s \succ u$  if and only if  $C \succ_C D$ .

Corollary 3.3:

The rewrite systems  $R_C$  and  $R_\infty$  are convergent (i. e., terminating and confluent).

# Superposition: Refutational Completeness

---

Lemma 3.4:

If  $D \preceq_C C$  and  $E_C = \{s \rightarrow t\}$ , then  $s \succ u$  for every term  $u$  occurring in a negative literal in  $D$  and  $s \preceq u$  for every term  $u$  occurring in a positive literal in  $D$ .

Corollary 3.5:

If  $D \in G_\Sigma(N)$  is true in  $R_D$ , then  $D$  is true in  $R_\infty$  and  $R_C$  for all  $C \succ_C D$ .

Corollary 3.6:

If  $D = D' \vee u \approx v$  is productive, then  $D'$  is false and  $D$  is true in  $R_\infty$  and  $R_C$  for all  $C \succ_C D$ .

# Superposition: Refutational Completeness

---

Lemma 3.7 (“Lifting Lemma”):

Let  $C$  be a clause and let  $\theta$  be a substitution such that  $C\theta$  is ground.

Then every equality resolution or equality factoring inference from  $C\theta$  is a ground instance of an inference from  $C$ .

Proof:

Exercise. □

# Superposition: Refutational Completeness

---

Lemma 3.8 (“Lifting Lemma”):

Let  $D = D' \vee u \approx v$  and  $C = C' \vee [\neg] s \approx t$  be two clauses (without common variables) and let  $\theta$  be a substitution such that  $D\theta$  and  $C\theta$  are ground.

If there is a superposition inference between  $D\theta$  and  $C\theta$  where  $u\theta$  and some subterm of  $s\theta$  are overlapped, and  $u\theta$  does not occur in  $s\theta$  at or below a variable position of  $s$ , then the inference is a ground instance of a superposition inference from  $D$  and  $C$ .

Proof:

Exercise. □

# Superposition: Refutational Completeness

---

Theorem 3.9 (“Model Construction”):

Let  $N$  be a set of clauses that is saturated up to redundancy and does not contain the empty clause. Then we have for every ground clause  $C\theta \in G_\Sigma(N)$ :

- (i)  $E_{C\theta} = \emptyset$  if and only if  $C\theta$  is true in  $R_{C\theta}$ .
- (ii) If  $C\theta$  is redundant w. r. t.  $G_\Sigma(N)$ , then it is true in  $R_{C\theta}$ .
- (iii)  $C\theta$  is true in  $R_\infty$  and in  $R_D$  for every  $D \in G_\Sigma(N)$  with  $D \succ_C C\theta$ .

## Superposition: Refutational Completeness

---

A  $\Sigma$ -interpretation  $\mathcal{A}$  is called **term-generated**, if for every  $b \in U_{\mathcal{A}}$  there is a ground term  $t \in T_{\Sigma}(\emptyset)$  such that  $b = \mathcal{A}(\beta)(t)$ .

Lemma 3.10:

Let  $N$  be a set of (universally quantified)  $\Sigma$ -clauses and let  $\mathcal{A}$  be a term-generated  $\Sigma$ -interpretation.

Then  $\mathcal{A}$  is a model of  $G_{\Sigma}(N)$  if and only if it is a model of  $N$ .

# Superposition: Refutational Completeness

---

Theorem 3.11 (Refutational Completeness: Static View):

Let  $N$  be a set of clauses that is saturated up to redundancy.

Then  $N$  has a model if and only if  $N$  does not contain the empty clause.

## Superposition: Refutational Completeness

---

So far, we have considered only inference rules that add new clauses to the current set of clauses

(corresponding to the *Deduce* rule of Knuth-Bendix Completion).

In other words, we have derivations of the form  $N_0 \vdash N_1 \vdash N_2 \vdash \dots$ , where each  $N_{i+1}$  is obtained from  $N_i$  by adding the consequence of some inference from clauses in  $N_i$ .

Under which circumstances are we allowed to delete (or simplify) a clause during the derivation?

# Superposition: Refutational Completeness

---

A **run** of the superposition calculus is a sequence

$N_0 \vdash N_1 \vdash N_2 \vdash \dots$ , such that

(i)  $N_i \models N_{i+1}$ , and

(ii) all clauses in  $N_i \setminus N_{i+1}$  are redundant w. r. t.  $N_{i+1}$ .

In other words, during a run we may add a new clause if it follows from the old ones, and we may delete a clause, if it is redundant w. r. t. the remaining ones.

For a run,  $N_\infty = \bigcup_{i \geq 0} \bigcap_{j \geq i} N_j$ .

The set  $N_\infty$  of all **persistent** clauses is called the **limit** of the run.

# Superposition: Refutational Completeness

---

Lemma 3.12:

If  $N \subseteq N'$ , then  $Red(N) \subseteq Red(N')$ .

Proof:

Obvious. □

# Superposition: Refutational Completeness

---

Lemma 3.13:

If  $N' \subseteq Red(N)$ , then  $Red(N) \subseteq Red(N \setminus N')$ .

Proof:

Like the corresponding Theorem 3.45 from the Automated Reasoning I lecture. □

# Superposition: Refutational Completeness

---

Lemma 3.14:

Let  $N_0 \vdash N_1 \vdash N_2 \vdash \dots$  be a run.

Then  $Red(N_i) \subseteq Red(\bigcup_{j \geq 0} N_j)$  and  $Red(N_i) \subseteq Red(N_\infty)$  for every  $i$ .

Proof:

Exercise. □

# Superposition: Refutational Completeness

---

Corollary 3.15:

$N_i \subseteq N_\infty \cup \text{Red}(N_\infty)$  for every  $i$ .

Proof:

If  $C \in N_i \setminus N_\infty$ , then there is a  $k \geq i$  such that  $C \in N_k \setminus N_{k+1}$ .

Therefore  $C$  must be redundant w. r. t.  $N_{k+1}$ .

Consequently,  $C$  is redundant w. r. t.  $N_\infty$ . □

# Superposition: Refutational Completeness

---

A run is called **fair**, if the conclusion of every inference from clauses in  $N_\infty \setminus Red(N_\infty)$  is contained in some  $N_i \cup Red(N_i)$ .

Lemma 3.16:

If a run is fair, then its limit is saturated up to redundancy.

Proof:

If the run is fair, then the conclusion of every inference from non-redundant clauses in  $N_\infty$  is contained in some  $N_i \cup Red(N_i)$ , and therefore contained in  $N_\infty \cup Red(N_\infty)$ .

Hence  $N_\infty$  is saturated up to redundancy. □

# Superposition: Refutational Completeness

---

Theorem 3.17 (Refutational Completeness: Dynamic View):

Let  $N_0 \vdash N_1 \vdash N_2 \vdash \dots$  be a fair run, let  $N_\infty$  be its limit.

Then  $N_0$  has a model if and only if  $\perp \notin N_\infty$ .

## 3.5 Improvements and Refinements

---

The superposition calculus as described so far can be improved and refined in several ways.

# Concrete Redundancy and Simplification Criteria

---

Redundancy is undecidable.

Even decidable approximations are often expensive  
(experimental evaluations are needed to see what pays off in practice).

Often a clause can be *made* redundant by adding another clause that is entailed by the existing ones.

This process is called **simplification**.

# Concrete Redundancy and Simplification Criteria

---

Examples:

Subsumption:

If  $N$  contains clauses  $D$  and  $C = C' \vee D\sigma$ , where  $C'$  is non-empty, then  $D$  subsumes  $C$  and  $C$  is redundant.

Example:

$f(x) \approx g(x)$  subsumes  $f(y) \approx a \vee f(h(y)) \approx g(h(y))$ .

# Concrete Redundancy and Simplification Criteria

---

Examples:

Trivial literal elimination:

Duplicated literals and trivially false literals can be deleted:

A clause  $C' \vee L \vee L$  can be simplified to  $C' \vee L$ ;

a clause  $C' \vee s \neq s$  can be simplified to  $C'$ .

# Concrete Redundancy and Simplification Criteria

---

Examples:

Condensation:

If we obtain a clause  $D$  from  $C$  by applying a substitution, followed by deletion of duplicated literals, and if  $D$  subsumes  $C$ , then  $C$  can be simplified to  $D$ .

Example:

By applying  $\{y \rightarrow g(x)\}$  to  $C = f(g(x)) \approx a \vee f(y) \approx a$  and deleting the duplicated literal, we obtain  $f(g(x)) \approx a$ , which subsumes  $C$ .

# Concrete Redundancy and Simplification Criteria

---

Examples:

Semantic tautology deletion:

Every clause that is a tautology is redundant. Note that in the non-equational case, a clause is a tautology if and only if it contains two complementary literals, whereas in the equational case we need a congruence closure algorithm to detect that a clause like  $x \neq y \vee f(x) \approx f(y)$  is tautological.

# Concrete Redundancy and Simplification Criteria

---

Examples:

Rewriting:

If  $N$  contains a unit clause  $D = s \approx t$  and a clause  $C[s\sigma]$ , such that  $s\sigma \succ t\sigma$  and  $C \succ_C D\sigma$ , then  $C$  can be simplified to  $C[t\sigma]$ .

Example:

If  $D = f(x, x) \approx g(x)$  and  $C = h(f(g(y), g(y))) \approx h(y)$ , and  $\succ$  is an LPO with  $h > f > g$ , then  $C$  can be simplified to  $h(g(g(y))) \approx h(y)$ .

## Redundant Inferences

---

So far, we have defined saturation in terms of redundant clauses:

$N$  is **saturated up to redundancy**, if the conclusion of every inference from clauses in  $N \setminus Red(N)$  is contained in  $N \cup Red(N)$ .

This definition ensures that in the proof of the model construction theorem, the conclusion  $C_0\theta$  of a ground inference follows from clauses in  $G_\Sigma(N)$  that are smaller than or equal to itself, hence they are smaller than the premise  $C\theta$  of the inference, hence they are true in  $R_{C\theta}$  by induction.

## Redundant Inferences

---

However, a closer inspection of the proof shows that it is actually sufficient that the clauses from which  $C_0\theta$  follows are smaller than  $C\theta$  – it is *not* necessary that they are smaller than  $C_0\theta$  itself.

This motivates the following definition of redundant *inferences*:

A ground inference with conclusion  $C_0$  and right (or only) premise  $C$  is called **redundant w. r. t. a set of ground clauses  $N$** , if one of its premises is redundant w. r. t.  $N$ , or if  $C_0$  follows from clauses in  $N$  that are smaller than  $C$ .

An inference is **redundant w. r. t. a set of clauses  $N$** , if all its ground instances are redundant w. r. t.  $G_\Sigma(N)$ .

## Redundant Inferences

---

Recall that a clause can be redundant w. r. t.  $N$  without being contained in  $N$ .

Analogously, an inference can be redundant w. r. t.  $N$  without being an inference from clauses in  $N$ .

The set of all inferences that are redundant w. r. t.  $N$  is denoted by  $RedInf(N)$ .

## Redundant Inferences

---

Saturation is then redefined in the following way:

$N$  is **saturated up to redundancy**, if every inference from clauses in  $N$  is redundant w. r. t.  $N$ .

Using this definition, the model construction theorem can be proved essentially as before.

## Redundant Inferences

---

The connection between redundant inferences and clauses is given by the following lemmas. They are proved in the same way as the corresponding lemmas for redundant clauses:

Lemma 3.18:

If  $N \subseteq N'$ , then  $RedInf(N) \subseteq RedInf(N')$ .

Lemma 3.19:

If  $N' \subseteq Red(N)$ , then  $RedInf(N) \subseteq RedInf(N \setminus N')$ .

# Selection Functions

---

Like the ordered resolution calculus, superposition can be used with a selection function that overrides the ordering restrictions for negative literals.

A **selection function** is a mapping

$$S : C \mapsto \text{set of occurrences of } \textit{negative} \text{ literals in } C$$

We indicate selected literals by a box:

$$\boxed{\neg f(x) \approx a} \vee g(x, y) \approx g(x, z)$$

# Selection Functions

---

The second ordering condition for inferences is replaced by

- The last literal in each premise is either selected, or there is no selected literal in the premise and the literal is maximal in the premise (strictly maximal for positive literals in superposition inferences).

In particular, clauses with selected literals can only be used in equality resolution inferences and as the second premise in negative superposition inferences.

# Selection Functions

---

Static refutational completeness is proved essentially as before:

We assume that each ground clause in  $G_{\Sigma}(N)$  inherits the selection of one of the clauses in  $N$  of which it is a ground instance (there may be several ones!).

In the proof of the model construction theorem, we replace case 3 by “ $C\theta$  contains a selected or maximal negative literal” and case 4 by “ $C\theta$  contains neither a selected nor a maximal negative literal”.

In addition, for the induction proof of this theorem we need one more property, namely:

(iv) If  $C\theta$  has selected literals then  $E_{C\theta} = \emptyset$ .

# Selection Functions

---

For dynamic refutational completeness, there is a problem, however:

In the static refutational completeness proof, the selection function  $gsel$  for ground clauses depends on the selection function  $sel$  for general clauses and on the saturated set  $N_\infty$  itself.

$N_\infty$  is the limit of a run, therefore it depends on  $RedInf$ .

$RedInf$  depends on what counts as a ground instance of an inference and what does not, and thus on the set of ground inferences.

The set of ground inferences depends of  $gsel$ , though!

How can we break this cycle?

# Selection Functions

---

Solution:

In the definition of *RedInf*, we have to quantify over all possible ground selection functions *gsel*:

An inference  $\iota$  is redundant, if for every ground selection function *gsel* corresponding to *sel*, all *gsel*-ground instances of  $\iota$  are redundant.

Result:

Worst-case analysis: When we check whether some inference involving a clause  $C \in N$  is redundant, we must assume that every ground instance  $D$  of  $C$  inherits the selection of  $C$  (even though  $D$  might also be a ground instance of another clause  $C' \in N$  with a different selection).

## 3.6 Splitting

---

Motivation:

A clause like  $f(x) \approx a \vee g(y) \approx b$  has rather undesirable properties in the superposition calculus: It does not have negative literals that one could select; it does not have a unique maximal literal; moreover, after performing a superposition inference with this clause, the conclusion often does not have a unique maximal literal either.

On the other hand, the two unit clauses  $f(x) \approx a$  and  $g(y) \approx b$  have much nicer properties.

## Splitting with Backtracking

---

If a clause  $\forall \vec{x}, \vec{y} C_1(\vec{x}) \vee C_2(\vec{y})$  consists of two non-empty variable-disjoint subclauses, then it is equivalent to the disjunction  $(\forall \vec{x} C_1(\vec{x})) \vee (\forall \vec{y} C_2(\vec{y}))$ .

In this case, superposition derivations can branch in a tableau-like manner:

Splitting: 
$$\frac{N \cup \{C_1 \vee C_2\}}{N \cup \{C_1\} \quad | \quad N \cup \{C_2\}}$$

where  $C_1$  and  $C_2$  do not have common variables.

If  $\perp$  is found on the left branch, backtrack to the right one.

# Splitting with Backtracking

---

If  $C_1$  is ground, the general rule can be improved:

Splitting: 
$$\frac{N \cup \{C_1 \vee C_2\}}{N \cup \{C_1\} \quad | \quad N \cup \{C_2\} \cup \{\neg C_1\}}$$

where  $C_1$  is ground.

Note:

$\neg C_1$  denotes the conjunction of all negations of literals in  $C_1$ .

## Splitting with Backtracking

---

In practice: most useful if both subclauses contain at least one positive literal.

## Implementing Splitting

---

Most clauses that are derived after a splitting step do *not* depend on the split clause.

It is unpractical to delete them as soon as one branch is closed and to recompute them in the other branch afterwards.

Solution: Associate a label set  $\mathcal{L}$  to every clause  $C$  that indicates on which splits it depends.

$$\text{Inferences: } \frac{C_2 \leftarrow \mathcal{L}_2 \quad C_1 \leftarrow \mathcal{L}_1}{C_0 \leftarrow \mathcal{L}_2 \cup \mathcal{L}_1}$$

# Implementing Splitting

---

If we derive  $\perp \leftarrow \mathcal{L}$  in one branch:

Determine the last split in  $\mathcal{L}$ .

Backtrack to the corresponding right branch.

Keep those clauses that are still valid on the right branch.

Restore clauses that have been simplified if the simplifying clause is no longer valid on the right branch.

Additionally: Delete splittings that did not contribute to the contradiction (branch condensation).

# AVATAR

---

Superposition with splitting has some similarity with CDCL.

Can we actually use CDCL?

# AVATAR

---

Encoding splitting components:

Use propositional literals as labels for splitting components:

non-ground component  $C \rightarrow$  prop. var.  $P_C$

positive ground component  $C \rightarrow$  prop. var.  $P_C$

negative ground component  $C \rightarrow$  negated prop. var.  $\neg P_C$

Therefore: splittable clauses  $\rightarrow$  propositional clauses.

# AVATAR

---

Implementation:

Combine a CDCL solver and a superposition prover.

The superposition prover passes splittable clauses and labelled empty clauses to the CDCL solver.

If the CDCL solver finds contradiction: input contradictory.

Otherwise the CDCL solver extracts a boolean model and passes the associated labelled clauses to the superposition prover.

## 3.7 Constraint Superposition

---

So far:

Refutational completeness proof for superposition is based on the analysis of inferences between ground instances of clauses.

Inferences between ground instances must be covered by inferences between original clauses.

Non-ground clauses represent the set of all their ground instances.

Do we really need *all* ground instances?

# Constrained Clauses

---

A **constrained clause** is a pair  $(C, K)$ , usually written as  $C \llbracket K \rrbracket$ , where  $C$  is a  $\Sigma$ -clause and  $K$  is a formula (called **constraint**).

Often:  $K$  is a boolean combination of *ordering literals*  $s \succ t$  with  $\Sigma$ -terms  $s, t$ .

(also possible: comparisons between literals or clauses).

Intuition:  $C \llbracket K \rrbracket$  represents the set of all ground clauses  $C\theta$  for which  $K\theta$  evaluates to true for some fixed term ordering. Such a  $C\theta$  is called a ground instance of  $C \llbracket K \rrbracket$ .

A clause  $C$  without constraint is identified with  $C \llbracket \top \rrbracket$ .

A constrained clause  $C \llbracket \perp \rrbracket$  with an unsatisfiable constraint represents no ground instances; it can be discarded.

# Constraint Superposition

---

Inference rules for constrained clauses:

Pos. Superposition: 
$$\frac{D' \vee t \approx t' \llbracket K_2 \rrbracket \quad C' \vee s[u] \approx s' \llbracket K_1 \rrbracket}{(D' \vee C' \vee s[t'] \approx s')\sigma \llbracket (K_2 \wedge K_1 \wedge K)\sigma \rrbracket}$$

where  $\sigma = \text{mgu}(t, u)$  and

$u$  is not a variable and

$$K = (t \succ t' \wedge s[u] \succ s' \\ \wedge (t \approx t') \succ_C D' \\ \wedge (s[u] \approx s') \succ_C C' \\ \wedge (s[u] \approx s') \succ_L (t \approx t'))$$

The other inference rules are modified analogously.

# Constraint Superposition

---

To work effectively with constrained clauses in a calculus, we need methods to check the satisfiability of constraints:

Possible for LPO, KBO, but expensive.

If constraints become too large, we may delete some conjuncts of the constraint. (Note that the calculus remains sound, if constraints are replaced by implied constraints.)

# Refutational Completeness

---

The refutational completeness proof for constraint superposition looks mostly like in Sect. 3.4.

Lifting works as before, so every ground inference that is required in the proof is an instance of some inference from the corresponding constrained clauses. (Easy.)

There is one significant problem, though.

## Refutational Completeness

---

Case 2 in the proof of Thm. 3.9 does not work for constrained clauses:

If we have a ground instance  $C\theta$  where  $x\theta$  is reducible by  $R_{C\theta}$ , we can no longer conclude that  $C\theta$  is true because it follows from some rule in  $R_{C\theta}$  and some smaller ground instance  $C\theta'$ .

Example: Let  $C \llbracket K \rrbracket$  be the clause  $f(x) \approx a \llbracket x \succ a \rrbracket$ , let  $\theta = \{x \mapsto b\}$ , and assume that  $R_{C\theta}$  contains the rule  $b \rightarrow a$ .

Then  $\theta$  satisfies  $K$ , but  $\theta' = \{x \mapsto a\}$  does not, so  $C\theta'$  is *not* a ground instance of  $C \llbracket K \rrbracket$ .

# Refutational Completeness

---

Solution:

Assumption: We start the saturation with a set  $N_0$  of *unconstrained* clauses; the limit  $N_\infty$  contains constrained clauses, though.

During the model construction, we ignore ground instances  $C\theta$  of clauses in  $N_\infty$  for which  $x\theta$  is reducible by  $R_{C\theta}$ .

## Refutational Completeness

---

We call a ground instance  $C\theta$  *variable irreducible* w. r. t. a ground TRS  $R$ , if for every variable  $x$  occurring in a literal  $L$  of  $C$ ,  $x\theta$  is irreducible by all rules in  $R$  that are smaller than  $L\theta$ .

The construction yields a TRS  $R_\infty$  that is a model of all  $R_\infty$ -*variable irreducible* ground instances of clauses in  $N_\infty$ .

$R_\infty$  is also a model of all  $R_\infty$ -*variable irreducible* ground instances of clauses in  $N_0$ .

## Refutational Completeness

---

Since all clauses in  $N_0$  are unconstrained, every ground instance of a clause in  $N_0$  follows from rules in  $R_\infty$  and some smaller or equal ground instance; so it is true in  $R_\infty$ .

Consequently,  $R_\infty$  is a model of *all* ground instances of clauses in  $N_0$ .

## Other Constraints

---

The approach also works for other kinds of constraints.

In particular, we can replace unification by equality constraints ( $\rightsquigarrow$  “basic superposition”):

Pos. Superposition: 
$$\frac{D' \vee t \approx t' \llbracket K_2 \rrbracket \quad C' \vee s[u] \approx s' \llbracket K_1 \rrbracket}{D' \vee C' \vee s[t'] \approx s' \llbracket K_2 \wedge K_1 \wedge K \rrbracket}$$

where  $u$  is not a variable and

$$K = (t = u)$$

Note: In contrast to ordering constraints, these constraints are essential for soundness.

## The Drawback

---

Constraints reduce the number of required inferences; however, they are detrimental to redundancy:

Since we consider only  $R_\infty$ -variable irreducible ground instances during the model construction, we may use only such instances for redundancy:

A clause is redundant, if all its  $R_\infty$ -variable irreducible ground instances follow from smaller  $R_\infty$ -variable irreducible ground instances and smaller rules in  $R_\infty$ .

Even worse, since we don't know  $R_\infty$  in advance, we must consider variable irreducibility w. r. t. arbitrary rewrite systems.

Consequence: Not every subsumed clause is redundant!

## 3.8 Hierarchic Superposition

---

The superposition calculus is a powerful tool to deal with formulas in *uninterpreted* first-order logic.

What can we do if some symbols have a *fixed interpretation*?

Can we combine superposition with decision procedures, e. g., for linear rational arithmetic?

Can we integrate the decision procedure as a “black box”?

## Sorted Logic

---

It is useful to treat this problem in sorted logic (cf. Sect. 1.11).

A many-sorted signature  $\Sigma = (\Xi, \Omega, \Pi)$  fixes an alphabet of non-logical symbols, where

- $\Xi$  is a set of sort symbols,
- $\Omega$  is a sets of function symbols,
- $\Pi$  is a set of predicate symbols.

## Sorted Logic

---

Each function symbol  $f \in \Omega$  has a unique declaration  $f : \xi_1 \times \cdots \times \xi_n \rightarrow \xi_0$ ; each predicate symbol  $P \in \Pi$  has a unique declaration  $P : \xi_1 \times \cdots \times \xi_n$  with  $\xi_i \in \Xi$ .

In addition, each variable  $x$  has a unique declaration  $x : \xi$ .

We assume that all terms, atoms, substitutions are well-sorted.

# Sorted Logic

---

A many-sorted algebra  $\mathcal{A}$  consists of

- a non-empty set  $\xi_{\mathcal{A}}$  for each  $\xi \in \Xi$ ,
- a function  $f_{\mathcal{A}} : \xi_{1,\mathcal{A}} \times \cdots \times \xi_{n,\mathcal{A}} \rightarrow \xi_{0,\mathcal{A}}$   
for each  $f : \xi_1 \times \cdots \times \xi_n \rightarrow \xi_0 \in \Omega$ ,
- a subset  $P_{\mathcal{A}} \subseteq \xi_{1,\mathcal{A}} \times \cdots \times \xi_{n,\mathcal{A}}$   
for each  $P : \xi_1 \times \cdots \times \xi_n \in \Pi$ .

# Hierarchic Specifications

---

A **specification**  $SP = (\Sigma, \mathcal{C})$  consists of

- a signature  $\Sigma = (\Xi, \Omega, \Pi)$ ,
- a class of term-generated  $\Sigma$ -algebras  $\mathcal{C}$  closed under isomorphisms.

If  $\mathcal{C}$  consists of *all* term-generated  $\Sigma$ -algebras satisfying the set of  $\Sigma$ -formulas  $N$ , we write  $SP = (\Sigma, N)$ .

# Hierarchic Specifications

---

A **hierarchic specification**  $HSP = (SP, SP')$  consists of

- a base specification  $SP = (\Sigma, \mathcal{C})$ ,
- an extension  $SP' = (\Sigma', N')$ ,

where  $\Sigma = (\Xi, \Omega, \Pi)$ ,  $\Sigma' = (\Xi', \Omega', \Pi')$ ,  
 $\Xi \subseteq \Xi'$ ,  $\Omega \subseteq \Omega'$ , and  $\Pi \subseteq \Pi'$ .

# Hierarchic Specifications

---

A  $\Sigma'$ -algebra  $\mathcal{A}$  is called a model of  $HSP = (SP, SP')$ ,  
if  $\mathcal{A}$  is a model of  $N'$  and  $\mathcal{A}|_{\Sigma} \in \mathcal{C}$ ,  
where the reduct  $\mathcal{A}|_{\Sigma}$  is defined as  $((\xi_{\mathcal{A}})_{\xi \in \Xi}, (f_{\mathcal{A}})_{f \in \Omega}, (P_{\mathcal{A}})_{P \in \Pi})$ .

Note:

- no confusion: models of  $HSP$  may not identify elements that are different in the base models.
- no junk: models of  $HSP$  may not add new elements to the interpretations of base sorts.

# Hierarchic Specifications

---

Example:

Base specification:  $((\Xi, \Omega, \Pi), \mathcal{C})$ , where

$$\Xi = \{int\}$$

$$\Omega = \{0, 1, -1, 2, -2, \dots : \rightarrow int, \\ - : int \rightarrow int, \\ + : int \times int \rightarrow int \}$$

$$\Pi = \{ \geq : int \times int, \\ > : int \times int \}$$

$\mathcal{C}$  = isomorphy class of  $\mathbb{Z}$

# Hierarchic Specifications

---

Example:

Extension:  $((\Xi', \Omega', \Pi'), N')$ , where

$$\Xi' = \Xi \cup \{list\}$$

$$\Omega' = \Omega \cup \{ \begin{array}{l} cons : int \times list \rightarrow list, \\ length : list \rightarrow int, \\ empty : \rightarrow list, \\ a : \rightarrow list \end{array} \}$$

$$\Pi' = \Pi$$

$$N' = \{ \begin{array}{l} length(a) \geq 1, \\ length(cons(x, y)) \approx length(y) + 1 \end{array} \}$$

# Hierarchic Specifications

---

Goal:

Check whether  $N'$  has a model in which the sort *int* is interpreted by  $\mathbb{Z}$  and the symbols from  $\Omega$  and  $\Pi$  accordingly.

# Hierarchic Superposition

---

In order to use a prover for the base theory, we must preprocess the clauses:

A term that consists only of base symbols and variables of base sort is called a base term (analogously for atoms, literals, clauses).

A clause  $C$  is called **weakly abstracted**, if every base term that occurs in  $C$  as a subterm of a non-base term (or non-base non-equational literal) is a variable.

Every clause can be transformed into an equivalent weakly abstracted clause. We assume that all input clauses are weakly abstracted.

# Hierarchic Superposition

---

A substitution is called simple, if it maps every variable of a base sort to a base term.

# Hierarchic Superposition

---

The inference rules of the hierarchic superposition calculus correspond to the rules of the standard superposition calculus with the following modifications:

- The term ordering  $\succ$  must have the property that every base ground term (or non-equational literal) is smaller than every non-base ground term (or non-equational literal).
- We consider only simple substitutions as unifiers.
- We perform only inferences on non-base terms (or non-base non-equational literals).
- If the conclusion of an inference is not weakly abstracted, we transform it into an equivalent weakly abstracted clause.

# Hierarchic Superposition

---

While clauses that contain non-base literals are manipulated using superposition rules, base clauses have to be passed to the base prover.

This yields one more inference rule:

Constraint Refutation:  $\frac{M}{\perp}$

where  $M$  is a set of base clauses that is inconsistent w. r. t.  $\mathcal{C}$ .

# Problems

---

There are two potential problems that are harmful to refutational completeness:

- We can only apply the constraint refutation rule to finite sets  $M$ . If  $\mathcal{C}$  is not compact, this is not sufficient.
- Since we only consider simple substitutions, we will only obtain a model of all *simple ground instances*.

To show that we have a model of *all* instances, we need an additional condition called *sufficient completeness w. r. t. simple instances*.

# Problems

---

A set  $N$  of clauses is called **sufficiently complete with respect to simple instances**, if for every model  $\mathcal{A}'$  of the set of simple ground instances of  $N$  and every ground non-base term  $t$  of a base sort there exists a ground base term  $t'$  such that  $t' \approx t$  is true in  $\mathcal{A}'$ .

Note: Sufficient completeness w. r. t. simple instances ensures the absence of junk.

# Problems

---

If the base signature contains Skolem constants, we can sometimes enforce sufficient completeness by equating ground extension terms with a base sort to Skolem constants.

Skolem constants may harmful to compactness, though.

# Completeness of Hierarchic Superposition

---

If the base theory is compact, the hierarchic superposition calculus is refutationally complete for sets of clauses that are sufficiently complete with respect to simple instances (Bachmair, Ganzinger, Waldmann, 1994; Baumgartner, Waldmann 2013).

Main proof idea: If the set of base clauses in  $N$  has some base model, represent this model by a set  $E$  of convergent ground equations and a set  $D$  of ground disequations.

Then show: If  $N$  is saturated w. r. t. hierarchic superposition, then  $E \cup D \cup \tilde{N}$  is saturated w. r. t. standard superposition, where  $\tilde{N}$  is the set of simple ground instances of clauses in  $N$  that are reduced w. r. t.  $E$ .

## A Refinement

---

In practice, a base signature often contains *domain elements*, that is, constant symbols that are

- guaranteed to be different from each other in every base model, and
- minimal w. r. t.  $\succ$  in their equivalent class.

Typical example for domain elements:  
number constants  $0, 1, -1, 2, -2, \dots$

## A Refinement

---

If the base signature contains *domain elements*, then weak abstraction can be redefined as follows:

A clause  $C$  is called **weakly abstracted**, if every base term that occurs in  $C$  as a subterm of a non-base term (or non-base non-equational literal) is a *variable or a domain element*.

Why does that work?

## 3.9 Integrating Theories I: E-Unification

---

Dealing with mathematical theories naively in a superposition prover is difficult:

Some axioms (e. g., commutativity) cannot be oriented w. r. t. a reduction ordering.

⇒ Provers compute many equivalent copies of a formula.

Some axiom sets (e. g., torsion-freeness, divisibility) are infinite.

⇒ Can we tell which axioms are really needed?

# Integrating Theories I: E-Unification

---

Hierarchic (“black-box”) superposition is easy to implement, but conditions like compactness and sufficient completeness are rather restrictive.

Can we integrate theories directly into theorem proving calculi (“white-box” integration)?

# Integrating Theories I: E-Unification

---

Idea:

In order to avoid enumerating entire congruence classes w. r. t. an equational theory  $E$ , treat formulas as *representatives* of their congruence classes.

Compute an inference between formula  $C$  and  $D$  if an inference between some clause represented by  $C$  and some clause represented by  $D$  would be possible.

Consequence: We have to check whether there are substitutions that make terms  $s$  and  $t$  equal w. r. t.  $E$ .

⇒ Unification is replaced by  $E$ -unification.

# E-Unification

---

**E-unification** (unification modulo an equational theory  $E$ ):

For a set of equality problems  $\{s_1 \approx t_1, \dots, s_n \approx t_n\}$ , an **E-unifier** is a substitution  $\sigma$  such that for all  $i \in \{1, \dots, n\}$ :  $s_i\sigma \approx_E t_i\sigma$ .

Recall:  $s_i\sigma \approx_E t_i\sigma$  means  $E \models s_i\sigma \approx t_i\sigma$ .

In general, there are infinitely many ( $E$ -)unifiers.

What about most general unifiers?

# E-Unification

---

Frequent cases:  $E = \emptyset$ ,  $E = AC$ ,  $E = ACU$ :

$$x + (y + z) \approx (x + y) + z \quad (\text{associativity} = A)$$

$$x + y \approx y + x \quad (\text{commutativity} = C)$$

$$x + 0 \approx x \quad (\text{identity (unit)} = U)$$

The identity axiom is also abbreviated by “1”, in particular, if the binary operation is denoted by  $*$ . ( $ACU = AC1$ ).

# E-Unification

---

Example:

$x + y$  and  $c$  are ACU-unifiable with  
 $\{x \mapsto c, y \mapsto 0\}$  and  $\{x \mapsto 0, y \mapsto c\}$ .

$x + y$  and  $x' + y'$  are ACU-unifiable with  
 $\{x \mapsto z_1 + z_2, y \mapsto z_3 + z_4, x' \mapsto z_1 + z_3, y' \mapsto z_2 + z_4\}$   
(among others).

# E-Unification

---

More general substitutions:

Let  $X$  be a set of variables.

A substitution  $\sigma$  is **more general modulo  $E$**  than a substitution  $\sigma'$  on  $X$ , if there exists a substitution  $\rho$  such that  $x\sigma\rho \approx_E x\sigma'$  for all  $x \in X$ .

Notation:  $\sigma \lesssim_E^X \sigma'$ .

(Why  $X$ ? Because we cannot restrict to substitutions that do not introduce new variables.)

# E-Unification

---

Complete sets of unifiers:

Let  $S$  be an  $E$ -unification problem, let  $X = \text{var}(S)$ .

A set  $C$  of  $E$ -unifiers of  $S$  is called **complete** (CSU), if for every  $E$ -unifier  $\sigma'$  of  $S$  there exists a  $\sigma \in C$  with  $\sigma \lesssim_E^X \sigma'$ .

A complete set of  $E$ -unifiers  $C$  is called **minimal** ( $\mu$ CSU), if for all  $\sigma, \sigma' \in C$ ,  $\sigma \lesssim_E^X \sigma'$  implies  $\sigma = \sigma'$ .

Note: every  $E$ -unification problem has a CSU. (Why?)

# E-Unification

---

The set of equations  $E$  is of unification type

**unitary**, if every  $E$ -unification problem has a  $\mu$ CSU with cardinality  $\leq 1$  (e. g.:  $E = \emptyset$ );

**finitary**, if every  $E$ -unification problem has a finite  $\mu$ CSU (e. g.:  $E = ACU$ ,  $E = AC$ ,  $E = C$ );

**infinitary**, if every  $E$ -unification problem has a  $\mu$ CSU and some  $E$ -unification problem has an infinite  $\mu$ CSU (e. g.:  $E = A$ );

**zero (or nullary)**, if some  $E$ -unification problem does not have a  $\mu$ CSU (e. g.:  $E = A \cup \{x + x \approx x\}$ ).

# Unification modulo ACU

---

Let us first consider **elementary ACU-unification**:

the terms to be unified contain only variables and the function symbols from  $\Sigma = (\{+/2, 0/0\}, \emptyset)$ .

Since parentheses and the order of summands don't matter, every term over  $X_n = \{x_1, \dots, x_n\}$  can be written as a sum  $\sum_{i=1}^n a_i x_i$ .

# Unification modulo ACU

---

The ACU-equivalence class of a term  $t = \sum_{i=1}^n a_i x_i \in T_{\Sigma}(X_n)$  is uniquely determined by the vector  $\vec{v}_n(t) = (a_1, \dots, a_n)$ .

Analogously, a substitution  $\sigma = \{ x_i \rightarrow \sum_{j=1}^m b_{ij} x_j \mid 1 \leq i \leq n \}$  is uniquely determined by the matrix

$$M_{n,m}(\sigma) = \begin{pmatrix} b_{11} & \cdots & b_{1m} \\ \vdots & & \vdots \\ b_{n1} & \cdots & b_{nm} \end{pmatrix}$$

# Unification modulo ACU

---

Let  $t = \sum_{i=1}^n a_i x_i$

and  $\sigma = \{ x_i \rightarrow \sum_{j=1}^m b_{ij} x_j \mid 1 \leq i \leq n \}$ .

$$\begin{aligned} \text{Then } t\sigma &= \sum_{i=1}^n a_i \left( \sum_{j=1}^m b_{ij} x_j \right) \\ &= \sum_{i=1}^n \sum_{j=1}^m a_i b_{ij} x_j \\ &= \sum_{j=1}^m \sum_{i=1}^n a_i b_{ij} x_j \\ &= \sum_{j=1}^m \left( \sum_{i=1}^n a_i b_{ij} \right) x_j. \end{aligned}$$

Consequence:

$$\vec{v}_m(t\sigma) = \vec{v}_n(t) \cdot M_{n,m}(\sigma).$$

# Unification modulo ACU

---

Let  $S = \{s_1 \approx t_1, \dots, s_k \approx t_k\}$  be a set of equality problems over  $T_\Sigma(X_n)$ .

Then the following properties are equivalent:

(a)  $\sigma$  is an ACU-unifier of  $S$  from  $X_n \rightarrow T_\Sigma(X_m)$ .

(b)  $\vec{v}_m(s_i\sigma) = \vec{v}_m(t_i\sigma)$  for all  $i \in \{1, \dots, k\}$ .

(c)  $\vec{v}_n(s_i) \cdot M_{n,m}(\sigma) = \vec{v}_n(t_i) \cdot M_{n,m}(\sigma)$  for all  $i \in \{1, \dots, k\}$ .

(d)  $(\vec{v}_n(s_i) - \vec{v}_n(t_i)) \cdot M_{n,m}(\sigma) = \vec{0}_m$  for all  $i \in \{1, \dots, k\}$ .

(e)  $M_{k,n}(S) \cdot M_{n,m}(\sigma) = \vec{0}_{k,m}$ .

where  $M_{k,n}(S)$  is the  $k \times n$  matrix whose rows are the vectors  $\vec{v}_n(s_i) - \vec{v}_n(t_i)$ .

# Unification modulo ACU

---

Then the following properties are equivalent (cont'd):

(e)  $M_{k,n}(S) \cdot M_{n,m}(\sigma) = \vec{0}_{k,m}$ .

where  $M_{k,n}(S)$  is the  $k \times n$  matrix whose rows are the vectors  $\vec{v}_n(s_i) - \vec{v}_n(t_i)$ .

(f) The columns of  $M_{n,m}(\sigma)$  are **non-negative integer solutions** of the system of **homogeneous linear diophantine equations**  $DE(S)$ :

$$M_{k,n}(S) \cdot \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

# Unification modulo ACU

---

Computing unifiers:

Obviously: if  $\vec{y}_1, \dots, \vec{y}_r$  are solutions of  $DE(S)$  and  $a_1, \dots, a_r$  are natural numbers, then  $a_1\vec{y}_1 + \dots + a_r\vec{y}_r$  is also a solution. (In particular, the zero vector is a solution!)

In fact, one can compute a **finite** set of solutions  $\vec{y}_1, \dots, \vec{y}_r$ , such that **every** solution of  $DE(S)$  can be represented as such a linear combination.

Moreover, if we combine these column vectors  $\vec{y}_1, \dots, \vec{y}_r$  to an  $n \times r$  matrix, this matrix represents a most general unifier of  $S$ . (Proof: see Baader/Nipkow.)

## From ACU to AC

---

A complete set of AC-unifiers for elementary AC-unification problems can be computed from a most general ACU-unifier by some postprocessing.

Elementary AC-unification is **finitary** and the elementary unifiability problem is solvable in polynomial time.

But that does not mean that minimal complete sets of AC-unifiers can be computed **efficiently**.

## From ACU to AC

---

E. Domenjoud has computed the exact size of AC- $\mu$ CSUs for unification problems of the following kind:

$$m x_1 + \cdots + m x_p \approx n y_1 + \cdots + n y_q$$

where  $\gcd(m, n) = 1$ .

The number of unifiers is

$$(-1)^{p+q} \sum_{i=0}^p \sum_{j=0}^q (-1)^{i+j} \binom{p}{i} \binom{q}{j} 2^{\binom{m+j-1}{m} \binom{n+i-1}{n}}$$

## From ACU to AC

---

For  $p = m = 1$  and  $q = n = 4$ , that is, for the equation

$$4x \approx y_1 + y_2 + y_3 + y_4$$

this is

34 359 607 481.

# From ACU to AC

---

Consequence:

If possible, avoid the **enumeration** of  $AC-\mu$ CSUs (which may have doubly exponential size).

Rather: only **check AC-unifiability**.

Or: **use ACU** instead.

# Unification with Constants

---

So far:

Elementary unification:

terms over variables and  $\{+, 0\}$  or  $\{+\}$ .

Step 2:

Additional **free constants**.

Step 3:

Additional **arbitrary free function symbols**.

$\rightsquigarrow$  Unification in the union of disjoint equational theories.

# Unification with Constants

---

Unification with constants:

We can treat constants  $a_i$  like variables  $x_i$  that *must* be mapped to themselves.

Consequence: The algorithm is similar to the one we have seen before, but we have to deal with homogeneous **and inhomogeneous** linear diophantine equations.

# Unification with Constants

---

Some complexity bounds change, however:

Unification type:

elementary ACU-unification: unitary;  
ACU-unification with constants: finitary.

Checking unifiability:

elementary ACU-unification: trivial;  
ACU-unification with constants: NP-complete.

# Combining Unification Procedures

---

The Baader–Schulz combination procedure allows to combine unification procedures for disjoint theories (e. g., ACU and the free theory).

Basic idea (as usual): Use abstraction to convert the combined unification problem into a union of two pure unification problems; solve them individually; combine the results.

# Combining Unification Procedures

---

Problem 1:

The individual unification procedures might map the same variable to different terms, e. g.,  $\{x \mapsto y + z\}$  and  $\{x \mapsto f(w)\}$ .

Solution: Guess for each variable non-deterministically which procedure treats it like a constant.

# Combining Unification Procedures

---

Problem 2:

Combining the results might produce cycles,  
e. g.,  $\{x \mapsto y + z\}$  and  $\{y \mapsto f(x)\}$ .

Solution: Guess an ordering of the variables non-deterministically; each individual unifier that is computed must respect the ordering.

Note: This is a non-trivial extension that may be impossible for some unification procedures

(but it is possible for *regular* equational theories, i. e., theories where for each equation  $u \approx v$  the terms  $u$  and  $v$  contain the same variables).

## 3.10 Integrating Theories II: Calculi

---

We can replace syntactic unification by  $E$ -unification in the superposition calculus.

Moreover, it is usually necessary to choose a term ordering in such a way that all terms in an  $E$ -congruence class behave in the same way in comparisons ( $E$ -compatible ordering).

However, this is usually not sufficient.

# AC and ACU

---

Example: Let  $E = AC$ . The clauses

$$a + b \approx d$$

$$b + c \approx e$$

$$c + d \not\approx a + e$$

are contradictory w. r. t. AC, but if  $a \succ b \succ c \succ d \succ e$ , then the maximal sides of these clauses are not AC-unifiable.

## AC and ACU

---

We have to compute inferences if some part of a maximal sum overlaps with a part of another maximal sum (the constant  $b$  in the example above).

Technically, we can do this in such a way that we first replace positive literals  $s \approx t$  by  $s + x \approx t + x$ , and then unify maximal sides w. r. t. AC or ACU (Peterson and Stickel 1981, Wertz 1992, Bachmair and Ganzinger 1994).

## AC and ACU

---

However, it turns out that even if we integrate AC or ACU in such a way into superposition, the resulting calculus is not particularly efficient – not even for ground formulas.

This is not surprising: The uniform word problem for AC or ACU is EXPSPACE-complete (Cardoza, Lipton, and Meyer 1976, Mayr and Meyer 1982).

# Abelian Groups

---

Working in Abelian groups is easier:

If we integrate also the inverse axiom, it is sufficient to compute inferences if **the maximal** part of a maximal sum overlaps with **the maximal** part of another maximal sum (like in Gaussian elimination).

Intuitively, in Abelian groups we can always isolate the maximal part of a sum on one side of an equation.

# Abelian Groups

---

What does that mean for the non-ground case?

Example:

$$g(y) + x \not\approx 2z \quad \vee \quad f(x) + z \approx 2y$$

Shielded variables ( $x, y$ ):

occur below a free function symbol,

$\rightsquigarrow$  cannot be mapped to a maximal term,

$\rightsquigarrow$  are not involved in inferences.

Unshielded variables ( $z$ ):

can be instantiated with  $m \cdot u + s$ , where  $u$  is maximal,

$\rightsquigarrow$  must be considered in inferences,

$\rightsquigarrow$  variable overlaps (similar to ACU).

# Abelian Groups

---

Variable overlaps are ugly:

If we want to derive a contradiction from

$$2a \approx c$$

$$2b \approx d$$

$$2x \not\approx c + d$$

and  $a \succ b \succ c \succ d$ , we have to map  $x$  to a sum of two variables  $x' + x''$ , unify  $x'$  with  $a$  and  $x''$  with  $b$ .

# Divisible Torsion-free Abelian Groups

---

Working in divisible torsion-free Abelian groups is still easier:

DTAGs permit variable elimination.

Every clause can be converted into a DTAG-equivalent clause without *unshielded* variables.

Since only overlaps of maximal parts of maximal sums have to be computed, variable overlaps become unnecessary.

Moreover, if abstraction is performed eagerly, terms to be unified do not contain  $+$ , so ACU-unification can be replaced by standard unification.

## Other Theories

---

A similar case: Chaining calculus for orderings.

$$\frac{D' \vee t' < t \quad C' \vee s < s'}{(D' \vee C' \vee t' < s')\sigma}$$

where  $\sigma$  is a most general unifier of  $t$  and  $s$ .

Avoids explicit inferences with transitivity.

Only maximal sides of ordering literals have to be overlapped.

But unshielded variables can be maximal.

In dense linear orderings without endpoints, all unshielded variables can be eliminated.

## Other Theories

---

DTAG-superposition and chaining can be combined to get a calculus for ordered divisible Abelian groups.

Again, all unshielded variables can be eliminated.

# Conclusion

---

Integrating theory axioms into superposition can become easier by integrating more axioms:

Easier unification problem ( $AC \rightarrow ACU$ ).

More restrictive inference rules ( $ACU \rightarrow AG$ ).

Fewer (or no) variable overlaps ( $AG \rightarrow DTAG$ ).

# Conclusion

---

Main drawback of all theory integration methods:

For each theory, we have to start from scratch, both for the completeness proof and the implementation.

## Part 4: Higher-Order Logic

---

Desired for applications, e. g., in mathematics:

quantifications over functions and predicates,

functions and predicates applied to functions and predicates,

partially applied functions,

anonymous functions,

first-class booleans,

expressivity: define, e. g., “the” natural numbers, “the” reals.

Higher-order logic satisfies these needs.

## 4.1 The Starting Point: $\lambda$ -Calculus

---

Untyped  $\lambda$ -calculus (Church 1930).

Syntax:

Terms: $t ::= c$	(Constant)
$x$	(Variable)
$(t_1 t_2)$	(Application)
$(\lambda x. t)$	(Abstraction)

# The Starting Point: $\lambda$ -Calculus

---

Substitution:

$$x\{x \mapsto s\} = s.$$

$$y\{x \mapsto s\} = y \text{ if } y \neq x.$$

$$c\{x \mapsto s\} = c.$$

$$(t_1 t_2)\{x \mapsto s\} = (t_1\{x \mapsto s\} t_2\{x \mapsto s\}).$$

$$(\lambda x. t)\{x \mapsto s\} = (\lambda x. t).$$

$$(\lambda y. t)\{x \mapsto s\} = (\lambda z. (t\{y \mapsto z\}\{x \mapsto s\})) \text{ if } y \neq x, z \text{ fresh.}$$

# The Starting Point: $\lambda$ -Calculus

---

Conversion rules (to be applied to arbitrary subterms):

$$t \rightarrow_{\alpha} t'$$

if  $t$  and  $t'$  are equal upto renaming of bound variables.

$$((\lambda x. t) s) \rightarrow_{\beta} t\{x \mapsto s\}.$$

$$(\lambda x. (t x)) \rightarrow_{\eta} t$$

if  $x$  does not occur freely in  $t$ .

# The Starting Point: $\lambda$ -Calculus

---

Properties of the untyped  $\lambda$ -calculus:

$\beta$ -conversion may not terminate.

Works as a model of computation (Turing-complete).

But basing a logic on it leads to problems  
(similarly to Russell's paradox).

Solution: introduce types.

## 4.2 Typed $\lambda$ -Calculus

---

Typed  $\lambda$ -calculus:

Developed by Church in 1940.

Also known as *Simple Type Theory*.

Note: Many variants (syntax and semantics).

# Types

---

Types are defined recursively:

$o$  is the type of Booleans; it is of order 0.

$\iota$  is the type of individuals; it is of order 1.

if  $\tau_1$  and  $\tau_2$  are types then  $\tau_1 \rightarrow \tau_2$  is a type;  
it is of order  $\max(\text{order}(\tau_1) + 1, \text{order}(\tau_2))$

We also write  $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$  or  $\tau_1, \dots, \tau_n \rightarrow \tau$  for  
 $\tau_1 \rightarrow (\dots \rightarrow (\tau_n \rightarrow \tau) \dots)$ .

# Terms

---

Given a non-empty set of constants and a collection of non-empty sets of variables for each type,

constants are terms,

variables are terms,

if  $t_1$  and  $t_2$  are terms then  $(t_1 t_2)$  is a term,

if  $x$  is a variable and  $t$  is a term then  $\lambda x. t$  is a term.

# Types of Terms

---

Given a non-empty set  $S$  of typed constants and a collection of non-empty sets of variables for each type, the term  $t$  is of type

$\perp$  if  $t \in \{\top, \perp\}$ ,

$\tau$  if  $t \in S$  has type  $\tau$ ,

$\tau$  if  $t = x_{(\tau)}$  is a variable of type  $\tau$ ,

$\tau_1 \rightarrow \tau_2$  if  $t = \lambda x_{(\tau_1)}. t_{1(\tau_2)}$ ,

$\tau_2$  if  $t = (t_{1(\tau_1 \rightarrow \tau_2)} t_{2(\tau_1)})$ .

A term is well-typed if a type can be associated to it according to the previous definition. We only consider well-typed terms in what follows.

# Normal Forms

---

For well-typed terms, we can define two kinds of normal forms:

$\beta\eta$ -short normal form:

Apply  $\beta$  forward exhaustively (terminates because of typing), then apply  $\eta$  forward exhaustively.

$\beta\eta$ -long normal form:

Apply  $\beta$  forward exhaustively (terminates because of typing), then apply  $\eta$  backward exhaustively (respecting the types).

# Extensions

---

Possible extensions:

Several types of individuals:  $c_1 : \iota_1, c_2 : \iota_2,$

Type constructors:  $x : list \iota,$

Polymorphic types:  $cons : \alpha \rightarrow list \alpha \rightarrow list \alpha,$

Dependent types:  $append : array \alpha n \rightarrow array \alpha m \rightarrow array \alpha (n + m)$

## 4.3 Semantics

---

A well-founded formula is a term of type  $o$ .

How to evaluate the truth of such a formula?

# Classical Models

---

Let  $D$  be a non-empty set, for each type  $\tau$  we define the following collection, denoted as the **frame** of the type

the frame of  $\tau = o$  is  $\llbracket o, D \rrbracket = \{0, 1\}$

the frame of  $\tau = \iota$  is  $\llbracket \iota, D \rrbracket = D$

the frame of  $\tau = \tau_1 \rightarrow \tau_2$  is  $\llbracket \tau_1 \rightarrow \tau_2, D \rrbracket$ , the collection of all functions mapping  $\llbracket \tau_1, D \rrbracket$  into  $\llbracket \tau_2, D \rrbracket$

# Classical Models

---

A higher-order **classical model** is a structure  $\mathcal{M} = \langle D, \mathcal{I} \rangle$  where  $D$  is a non-empty set called the **domain** of the model and  $\mathcal{I}$  is the **interpretation** of the model, a mapping such that

if  $c_{(\tau)}$  is a constant then  $\mathcal{I}(c) \in \llbracket \tau, D \rrbracket$ ,

$\mathcal{I}(=_{(\tau \rightarrow \tau \rightarrow o)})$  is the equality relation on  $\llbracket \tau, D \rrbracket$ .

By adding an **assignment** function  $\alpha$  such that for any variable  $x_{(\tau)}$ ,  $\alpha(x) \in \llbracket \tau, D \rrbracket$ , it becomes possible to evaluate the truth value of higher-order formulas as in first-order logic.

# Classical Models

---

The **evaluation**  $\mathcal{V}_{\mathcal{M},\alpha}(t)$  of a term  $t$  given a model  $\mathcal{M} = \langle D, \mathcal{I} \rangle$  and an assignment  $\alpha$  is recursively defined as

$\mathcal{I}(c)$  if  $t$  is a constant  $c$

$\alpha(x)$  if  $t$  is a variable  $x$

the function from  $[[\tau_1, D]]$  to  $[[\tau_2, D]]$  that maps every  $a \in [[\tau_1, D]]$  to

$\mathcal{V}_{\mathcal{M},\alpha[x \mapsto a]}(t) \in [[\tau_2, D]]$  if  $t = \lambda x_{(\tau_1)}. t_{(\tau_2)}$

$(\mathcal{V}_{\mathcal{M},\alpha}(t_1))(\mathcal{V}_{\mathcal{M},\alpha}(t_2))$  if  $t = (t_{1(\tau_1 \rightarrow \tau_2)} t_{2(\tau_1)})$

# Classical Models

---

Truth evaluation:

Given a model  $\mathcal{M} = \langle D, \mathcal{I} \rangle$  and an assignment  $\alpha$ , a well-founded formula  $F$  is true in  $\mathcal{M}$  with respect to  $\alpha$ , denoted as  $\mathcal{M}, \alpha \models F$  iff  $\mathcal{V}_{\mathcal{M}, \alpha}(F) = 1$

$F$  is satisfiable in  $\mathcal{M}$  iff there exists an assignment  $\alpha$  such that  $\mathcal{M}, \alpha \models F$

$F$  is valid in  $\mathcal{M}$ , denoted  $\mathcal{M} \models F$  iff for all assignments  $\alpha$ ,  $\mathcal{M}, \alpha \models F$

$F$  is valid, denoted  $\models F$  iff for all models  $\mathcal{M}$ ,  $\mathcal{M} \models F$

These notions extend straightforwardly to sets of formulas.

# Problems with the Classical Semantics

---

HOL with classical semantics (cHOL) is very expressive, but:

- In FOL, every unsatisfiable set of formulas has a finite unsatisfiable subset. This is no longer the case in cHOL. (Loss of compactness)
- No proof procedure able to derive all consequences of a set of formulas can exist in cHOL. (Loss of strong completeness)
- No proof procedure able to derive all valid sets of formulas can exist in cHOL. (Loss of weak completeness)
- The status of validity of some formulas is unclear.

# Henkin Semantics

---

To solve the previously mentioned issues, it is possible to generalize the notion of a model by relaxing the notion of a frame into that of a Henkin frame. Given a non-empty set  $D$ ,

$$\llbracket o, D \rrbracket = \{0, 1\}$$

$$\llbracket \iota, D \rrbracket = D$$

$\llbracket \tau_1 \rightarrow \tau_2, D \rrbracket$  is ~~the~~ **some** collection of ~~all~~ functions mapping  $\llbracket \tau_1, D \rrbracket$  into  $\llbracket \tau_2, D \rrbracket$  **with some additional closure conditions.**

# Henkin vs Classical Semantics

---

- Every classical model is a Henkin model, therefore every formula true in all Henkin models is true in all classical models.
- There are formulas true in all classical models that are not true in all Henkin models.
- There are (weak) complete proof procedures for HOL with Henkin semantics.

## 4.4 Higher-Order Term Unification

---

We consider unification modulo  $\alpha, \beta, \eta$ .

In FOL, there exists a unique mgu for two unifiable terms.

This is no longer true in HOL.

For example, consider  $t_1 = y x$  and  $t_2 = c$  where  $x, y$  are variables and  $c$  is a constant. The unifiers of  $t_1$  and  $t_2$  are  $\{y \mapsto \lambda z. c\}$  and  $\{y \mapsto \lambda z. z, x \mapsto c\}$ .

Some equations do not even have finite complete sets of unifiers, e. g.,  $f (x c) = x (f c)$  with  $f$  and  $c$  constants.

Even worse, the higher-order unification problem is *undecidable*.

# Huet's Unification Algorithm

---

Given:

a unification problem  $E$ , i.e. a finite set of equations in  $\beta$  and, on the outermost level, in  $\eta$ -long normal form.

Goal:

find a substitution  $\sigma$  such that  $E\sigma$  contains only syntactically equal equations.

## Rigid and Flexible Terms

---

Every term can be written in the form  $\lambda x_1 \dots x_n. u_0 u_1 \dots u_k$  ( $n \geq 0$ ,  $k \geq 0$ ), where  $u_0$  is a constant or a bound or free variable.

$u_0$  is called the *head* of the term.

A term is called **rigid** if its head symbol is a constant or a bound variable. Otherwise its head symbol is a free variable and the term is called **flexible**.

## Rigid-Rigid Equations

---

Two rules can be applied depending on the head symbols in the rigid-rigid equation.

Simplify:

$$\frac{E \cup \{\lambda x_1 \dots x_n. f \ u_1 \dots u_p \approx \lambda x_1 \dots x_n. f \ v_1 \dots v_p\}}{E \cup \{\lambda x_1 \dots x_n. u_1 \approx \lambda x_1 \dots x_n. v_1, \dots, \lambda x_1 \dots x_n. u_p \approx \lambda x_1 \dots x_n. v_p\}}$$

where  $f$  is a constant or a bound variable.

# Rigid-Rigid Equations

---

Fail:

$$\frac{E \cup \{\lambda x_1 \dots x_n. f \ u_1 \dots u_p \approx \lambda x_1 \dots x_n. g \ v_1 \dots v_q\}}{\perp}$$

where  $f$  and  $g$  are distinct constants or bound variables.

## Flexible-Rigid Equations

---

There is only one rule to handle such equations, but it is a branching rule.

Generate:

$$\frac{E}{E\sigma}$$

where

$$(\lambda x_1 \dots x_n. z u_1 \dots u_p \approx \lambda x_1 \dots x_n. f v_1 \dots v_q) \in E,$$

$z$  is a free variable,  $f$  is a constant or bound variable,

$h \in \{f, y_1, \dots, y_p\}$  if  $f$  is a constant and  $h \in \{y_1, \dots, y_p\}$  otherwise,

$z_1, \dots, z_r$  are fresh free variables,

$$\sigma = \{z \mapsto \lambda y_1, \dots, y_p. h (z_1 y_1 \dots y_p) \dots (z_r y_1 \dots y_p)\}.$$

## Flexible-Flexible Equations

---

The following result, also by Huet, handles flexible-flexible equations.

Proposition 4.1:

A unification problem  $E$  containing only flexible-flexible equations has always a solution.

We say that a unification problem with only flexible-flexible equations is a **solved** unification problem.

## The Whole Procedure

---

A reasonable strategy consists in applying Fail and Simplify eagerly, and Generate only when there is no rigid-rigid equation left.

Generate is non-deterministic, making this procedure branching.

# Soundness and Completeness

---

Proposition 4.2:

If a unification problem  $E$  can be transformed into a solved problem  $E'$  by applying Fail, Simplify and Generate then  $E$  has a solution.

Proposition 4.3:

If a unification problem  $E$  has a solution  $\sigma$  then we can derive a solved problem  $E'$  from  $E$  using the rules Fail, Simplify and Generate.

Theorem 4.4:

The procedure made of the rules Fail, Simplify, and Generate is sound and complete.

# Termination?

---

Higher-order unification is only semi-decidable.

When solutions exist, Huet's algorithm will find one and terminate, but when there is no solution, it may loop forever.

## Alternatives

---

Huet's procedure tests only unifiability.

There are also unification procedures for higher-order logic that enumerate a complete set of unifiers, e.g., by Jensen and Pietrzykowski and by Vukmirović et al.

## 4.5 Resolution in Higher-Order Logic

---

In first-order logic, resolution for general clauses has two rules:

$$\textit{Resolution:} \quad \frac{D \vee B \quad C \vee \neg A}{(D \vee C)\sigma}$$

where  $\sigma = \text{mgu}(A, B)$ .

$$\textit{Factoring:} \quad \frac{C \vee A \vee B}{(C \vee A)\sigma}$$

where  $\sigma = \text{mgu}(A, B)$ .

# Resolution in Higher-Order Logic

---

In higher-order logic, a first problem is that mgu's need not exist and unification is undecidable.

Example 4.5:

Given  $D \vee B$  and  $C \vee \neg A$  where  $A$  and  $B$  are unifiable but without mgu, there may exist infinitely many  $\sigma_1, \sigma_2, \dots$  unifiers of  $A$  and  $B$  generating distinct resolvents  $(D \vee C)\sigma_1, (D \vee C)\sigma_2, \dots$  and in general there is no way to know which one is needed to prove the given theorem.

# Resolution in Higher-Order Logic

---

Huet proposes to delay the computation of unifiers (when no mgu exists) by using constraints storing the corresponding unification problems.

Once a contradiction has been derived, the corresponding unification problem can then be solved using Huet's algorithm.

$$\textit{Resolution:} \quad \frac{D \vee B[X] \quad C \vee \neg A[Y]}{D \vee C[X \wedge Y \wedge A = B]}$$

$$\textit{Factoring:} \quad \frac{C \vee A \vee B[X]}{C \vee A[X \wedge A = B]}$$

## Resolution in Higher-Order Logic

---

Another problem in HOL is that it is not always possible to guess the necessary substitution based on the available terms.

Example 4.6:

Consider the formula  $\neg X_{(o)}$  where  $X$  is a Boolean variable. The set  $\{\neg X\}$  is saturated by resolution, but still the formula  $\neg X$  is unsatisfiable. However, we can guess the substitution  $\sigma = \{X \mapsto \neg Y\}$ . Then  $(\neg X)\sigma = \neg(\neg Y) = Y$  and resolution can now derive the empty clause from  $\neg X$  and  $Y$ .

# Resolution in Higher-Order Logic

---

To overcome this issue, Huet introduces additional **splitting rules**.

$$\frac{C \vee A[X]}{C \vee \neg x_{(o)} [X \wedge A = \neg x]}$$

$$\frac{C \vee A[X]}{C \vee x_{(o)} \vee y_{(o)} [X \wedge A = (x \vee y)]}$$

$$\frac{C \vee A[X]}{C \vee P_{(\tau \rightarrow o)} x_{(\tau)} [X \wedge A = \Pi_{((\tau \rightarrow o) \rightarrow o)} P]}$$

$\Pi_{((\tau \rightarrow o) \rightarrow o)}$  is the function that associates  $\top$  to any set of type  $\tau \rightarrow o$  that contains all elements of type  $\tau$ .

# Resolution in Higher-Order Logic

---

$$\frac{C \vee \neg A[X]}{C \vee x_{(o)} [X \wedge A = \neg x]}$$

$$\frac{C \vee \neg A[X]}{C \vee \neg x_{(o)} [X \wedge A = (x \vee y_{(o)})], \quad C \vee \neg y [X \wedge A = (x \vee y)]}$$

$$\frac{C \vee \neg A[X]}{C \vee \neg P_{(\tau \rightarrow o)}(\text{sk}_{((\tau \rightarrow o) \rightarrow \tau)} P) [X \wedge A = \Pi_{((\tau \rightarrow o) \rightarrow o)} P]}$$

sk is the Skolem constant such that  $\neg \Pi_{((\tau \rightarrow o) \rightarrow o)} P = \neg P_{(\tau \rightarrow o)}(\text{sk}_{((\tau \rightarrow o) \rightarrow \tau)} P)$ .

# Resolution in Higher-Order Logic

---

Resolution with these splitting rules is sound and complete, but not terminating (Huet).

# Resolution in Higher-Order Logic

---

In practice, several improvements are possible.

As soon as a constraint becomes unsatisfiable, delete the corresponding clause.

If a constraint has a small enough set of solutions, generate all applied clauses to replace the constrained original one.

## 4.6 Superposition in Higher-Order Logic

---

Problems originating from proof assistants

use higher-order logic,

but contain large first-order parts,

and in particular equality.

Can we extend the superposition calculus to higher-order logic?

Following Bentkamp et al. we proceed in three steps.

# Step 1: Lambda-free HOL

---

We admit

- applied variables ( $x\ b\ c$ ),
- unapplied or partially applied functions ( $g\ f\ \approx\ h\ (f\ b\ c)$ )

but exclude

- lambda abstractions
- first-class booleans (i. e., boolean expressions on the term level, rather than on the literal level)

## Step 1: Lambda-free HOL

---

This is also known as the **applicative fragment**.

In principle, one could encode it in FOL using constants and just one binary function symbol *app*.

FOL provers do not behave well on these formulas, though:

Indexing data structures become almost useless.

Term orderings do not behave in the expected way anymore.

# Step 1: Lambda-free HOL

---

First step:

Define a higher-order reduction ordering.

In addition to the usual properties of reduction orderings, one would like to have **compatibility with arguments**:  $s \succ s'$  implies  $s t \succ s' t$ .

But this is difficult to achieve.

The calculus below works without this requirement.

## Step 1: Lambda-free HOL

---

A subterm  $t$  of  $s$  is called a **green subterm**, if  $t = s$  or if  $s = s' u_1 \dots u_n$  and  $t$  is a green subterm of some  $u_i$ .

Notation:  $s = s\langle t \rangle$ .

Green subterms correspond to first-order subterms.

If  $t \succ t'$ , then  $s\langle t \rangle \succ s\langle t' \rangle$ .

# Step 1: Lambda-free HOL

---

Second step:

Define the inference rules analogously to first-order superposition, but restrict to inferences that involve green subterms:

$$\frac{D' \vee t \approx t' \quad C' \vee s\langle u \rangle \approx s'}{(D' \vee C' \vee s\langle t' \rangle \approx s')\sigma}$$

where  $\sigma = \text{mgu}(t, u)$ .

# Step 1: Lambda-free HOL

---

Additionally:

New inference rule: ArgCong

$$\frac{C' \vee s \approx s'}{C' \vee s x \approx s' x}$$

Redundancy for inferences must be defined in such a way that the conclusion of ArgCong is not automatically redundant!

## Step 1: Lambda-free HOL

---

One more problem:

If  $\succ$  is not compatible with arguments, we need occasionally superpositions at (but not below) variable positions.

(The  $\theta/\theta'$  trick may not work anymore.)

# Step 1: Lambda-free HOL

---

Proof idea:

Use a two-fold lifting

- from HOL to ground HOL
- from ground HOL to ground FOL

## Step 1: Lambda-free HOL

---

Note: The Henkin interpretations contain only those functions that we can construct from the given ones.

In order to refute  $b \neq x b$ , our set of axioms should contain  $id\ z \approx z$ .

## Step 2: Boolean-free HOL

---

We add lambda abstractions to the logic,  
but still exclude first-class booleans.

## Step 2: Boolean-free HOL

---

Need efficient HO unification procedure that enumerates a CSU (Vukmirović et al.)

Need dovetailing to interleave generation of further conclusions of inferences with clause selection.

## Step 2: Boolean-free HOL

---

Again: only inferences involving green subterms.

The definition of green subterms must be adapted, though:

A subterm  $t$  of  $s$  is called a **green subterm**, if  $t = s$  or if  $s = c u_1 \dots u_n$  for some constant  $c$  and  $t$  is a green subterm of some  $u_i$ .

(Subterms below applied variables are no longer green.)

## Step 2: Boolean-free HOL

---

Problem: Applying a grounding substitution  $\theta$  that maps free variables to lambda expressions may fundamentally change the structure of a term  $t$ . Green subterms in  $t\theta$  need not be instances of green subterms in  $t$ .

Examples:

Let  $t = h (x b f)$  and  $\theta = \{x \mapsto \lambda y z. g (z y)\}$ .

Then  $t\theta = h (g (f b))$  contains the green subterm  $f b$ .

Let  $t = \lambda x. f (y x) x$  and  $\theta = \{y \mapsto \lambda z. c\}$ .

Then  $t\theta = \lambda x. f c x = f c$  contains the green subterm  $f c$ .

## Step 2: Boolean-free HOL

---

Solution:

Need fluid inferences to ensure that all inferences between ground instances can be lifted:

$$\frac{D' \vee t \approx t' \quad C' \vee s\langle u \rangle \approx s'}{(D' \vee C' \vee s\langle z t' \rangle \approx s')\sigma}$$

where  $\sigma \in \text{CSU}(z t, u)$ .

## Step 3: Full HOL

---

We add first-class booleans to the logic.

## Step 3: Full HOL

---

New problem: Performing a CNF transformation (including Skolemization) a priori is no longer sufficient.

Solution: Construct the HO superposition calculus on top of a **non-clausal** FO superposition calculus that performs CNF transformation steps on the fly.

## Step 3: Full HOL

---

We inherit the **hoisting** inferences of the non-clausal FO superposition calculus, e. g.

$$\frac{C\langle u \rangle}{(C\langle \perp \rangle \vee x \approx y)\sigma}$$

where  $\sigma \in \text{CSU}(u, x \approx y)$ .

$$\frac{C\langle u \rangle}{(C\langle \top \rangle \vee x \approx y)\sigma}$$

where  $\sigma \in \text{CSU}(u, x \not\approx y)$ .

## Step 3: Full HOL

---

Implementations:

Zipperposition (OCaml, full calculus)

E (C, parts of the calculus)

## Alternative Approach

---

Extending an existing prover for FOL to handle lambda expressions requires substantial modifications of the architecture.

Alternative approach (Bhayat and Reger):

Every lambda expression can be encoded using a small set of **combinators**, e. g.,  $S = \lambda x y z. x z (y z)$ ,  $K = \lambda x y. x$ ,  $I = \lambda x. x$ .

Use the lambda-free calculus together with the definitions of combinators.

Implemented in Vampire.