Natty:
A Natural-Language Proof Assistant for Higher-Order Logic

Adam Dingle

Charles University

Sep 5, 2024

Introduction: a question

- Can current systems automatically verify proof steps in textbook mathematics almost all of the time?
  - If so, formalizing mathematics should be (relatively) easy
  - If not, why not?

Natty

- Natty: a new natural-language proof assistant
- User writes axioms/theorems/proofs in (controlled) natural language
- Natty translates them into higher-order logic
- ...and formally proves that they are true

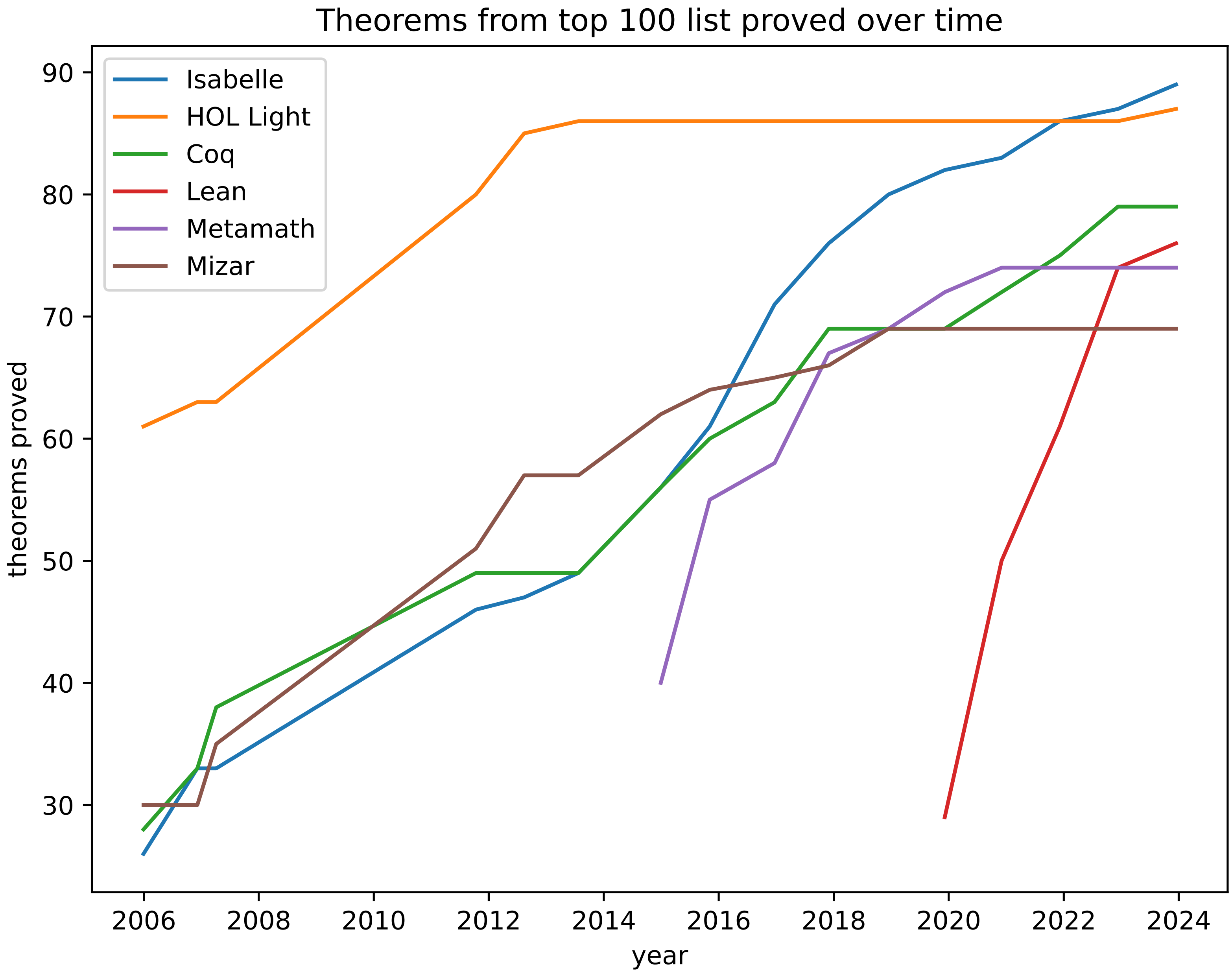
Natty: a nascent project

- Initial commit on Feb 18, 2024
- About 3,200 lines of OCaml code
- Work in progress!
- Today, can only prove some statements about $\mathbb{N}$ and $\mathbb{Z}$
- Goal: expand to general mathematics
- Online: https://github.com/medovina/natty

A benchmark: Wiedijk's 100 theorems

1. The Irrationality of the Square Root of 2
2. Fundamental Theorem of Algebra
3. The Denumerability of the Rational Numbers
4. $\cdots$

A benchmark: Wiedijk’s 100 theorems



nat.n - prover - Visual Studio Code

nat.n

Natural numbers: definition

Axiom 1. There exists a type $\mathbb{N}$ with an element $0 \in \mathbb{N}$ and a function $s : \mathbb{N} \to \mathbb{N}$ such that

a. There is no $n \in \mathbb{N}$ such that $s(n) = 0$.
b. For all $n, m \in \mathbb{N}$, if $s(n) = s(m)$ then $n = m$.
c. Let $P : \mathbb{N} \to \mathbb{B}$. If $P(0)$ is true, and $P(k)$ implies $P(s(k))$ for all $k \in \mathbb{N}$, then $P(n)$ is true for all $n \in \mathbb{N}$.

Definition. Let $1 : \mathbb{N} = s(0)$.

Lemma 1. Let $a \in \mathbb{N}$. Suppose that $a \neq 0$. Then there is some $b \in \mathbb{N}$ such that $a = s(b)$.

Natural numbers: addition

Axiom 2. There is a binary operation $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that for all $n, m \in \mathbb{N}$,

a. $n + 0 = n$.
b. $n + s(m) = s(n + m)$.

Theorem 2. Let $a, b, c \in \mathbb{N}$.

1. If $a + c = b + c$, then $a = b$.
2. $(a + b) + c = a + (b + c)$.
3. $0 + a = a = a + 0$.
4. $s(a) + b = s(a + b)$.
5. $a + b = b + a$.
6. $a + s(b) \neq 0$.
7. $a + s(b) \neq a$.
8. If $a + b = 0$, then $a = 0$ and $b = 0$.

main ⊗ 0 ⚠ 0 opam(default) Ln 29, Col 41 Spaces: 2 UTF-8 LF Natty Proving: off

nat.n - prover - Visual Studio Code

nat.n

Axiom 3. There is a binary operation $\cdot : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ such that for all $n, m \in \mathbb{N}$,

a. $n \cdot 0 = 0$.
b. $n \cdot s(m) = (n \cdot m) + n$.

Theorem 3. Let $a, b, c \in \mathbb{N}$.

1. $a \cdot 0 = 0 = 0 \cdot a$.
2. $a \cdot 1 = a = 1 \cdot a$.
3. $c(a + b) = ca + cb$.
4. $(ab)c = a(bc)$.
5. $s(a) \cdot b = ab + b$.
6. $ab = ba$.
7. $(a + b)c = ac + bc$.

Proof.

4. Let $a, b \in \mathbb{N}$. Let

$$G = \{ x \in \mathbb{N} \mid (a \cdot b) \cdot x = a \cdot (b \cdot x) \}.$$

We have $(ab) \cdot 0 = 0 = a(b \cdot 0)$, so $0 \in G$. Now let $c \in \mathbb{N}$, and suppose that $c \in G$. Then

$$\begin{aligned} (a \cdot b) \cdot s(c) &= (a \cdot b) \cdot c + a \cdot b && \text{by Axiom 3(b)} \\ &= a \cdot (b \cdot c) + a \cdot b \\ &= a \cdot (b \cdot c + b) && \text{by Theorem 3.3} \\ &= a \cdot (b \cdot s(c)) && \text{by Axiom 3(b).} \end{aligned}$$

So $s(c) \in G$. Hence $x \in G$ for all $x \in \mathbb{N}$.

main 0 0 opam(default) Ln 38, Col 29 Spaces: 2 UTF-8 LF Natty Proving: off

nat.n - prover - Visual Studio Code

nat.n

Theorem 7. Let $a, b \in \mathbb{N}$.

1. Precisely one of $a < b$ or $a = b$ or $a > b$ holds.
2. $a \leq b$ or $b \leq a$.
3. If $a \leq b$ and $b \leq a$, then $a = b$.
4. It cannot be that $b < a < b + 1$.
5. $a \leq b$ if and only if $a < b + 1$.
6. $a < b$ if and only if $a + 1 \leq b$.

Proof.

1. Let $a, b \in \mathbb{N}$. Suppose that $a < b$ and $a = b$. It then follows that $a < a$, which is a contradiction to Theorem 6.1. Now suppose that $a = b$ and $a > b$. It follows that $a > a$, which is similarly a contradiction. Now suppose that $a < b$ and $b < a$. By Theorem 6.6 we deduce that $a < a$, again a contradiction.

Now let

$$G = \{ x \in \mathbb{N} \mid \text{for all } y \in \mathbb{N}, x < y \text{ or } x = y \text{ or } x > y \}.$$

We will show that $x \in G$ for all $x \in \mathbb{N}$. We start by showing that $0 \in G$. Let $y \in \mathbb{N}$. By Theorem 6.2 we know that $0 \leq y$. It follows that $y = 0$ or $y > 0$. Hence $0 \in G$.

Now let $x \in \mathbb{N}$, and suppose that $x \in G$. We will show that $s(x) \in G$. Let $y \in \mathbb{N}$. By hypothesis we know that $x < y$ or $x = y$ or $x > y$.

First suppose that $x < y$. Then there is some $p \in \mathbb{N}$ such that $x + p = y$. If $p = 0$, then $x = y$, so $s(x) > y$ by Theorem 6.1. If $p \neq 0$, then by Lemma 1 there is some $r \in \mathbb{N}$ such that $p = s(r)$, which implies that $x + s(r) = y$, which implies $s(x) + r = y$, so $s(x) \leq y$, so either $s(x) < y$ or $s(x) = y$.

Next suppose that $x = y$. Then by Theorem 6.1 it follows that $s(x) > x = y$. Finally, suppose that $x > y$. We know that $s(x) > x$, and by Theorem 6.6 it follows that $s(x) > y$.

Putting the cases together, we see that $s(x) < y$ or $s(x) = y$ or $s(x) > y$ always holds. Hence $s(x) \in G$, and we conclude that $x \in G$ for all $x \in \mathbb{N}$.

main ⊗ 0 ⚠ 0 opam(default) Ln 43, Col 21 Spaces: 2 UTF-8 LF Natty Proving: off

Input language

- Axioms, definitions, lemmas/theorems, proofs
- Implicit multiplication
- User must specify a type for every variable
- Supports set comprehension syntax
  - a set is a function with codomain $\mathbb{B}$
- Type overloading
  - $0 : \mathbb{N}$ and $0 : \mathbb{Z}$
  - $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and $+ : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$
- No polymorphism (yet)!
- Proof steps may invoke a previous lemma/theorem

Input file: nat.n

- Defines ℕ axiomatically (Peano axioms)
  - Defines +, ·, < axiomatically
  - Using axioms for definitions is not great - this will change
- 37 theorems about ℕ
  - 9 with hand-written proofs
  - 102 proof steps
- Defines ℤ axiomatically
  - Isomorphic to equivalence class of (ℕ, ℕ)
- Defines +, −, ·, < on ℤ axiomatically
- 22 theorems about ℤ
  - 12 with hand-written proofs
  - 106 proof steps

Running Natty

- Can run from command line, or interactively via VS Code extension
- Output: THF file for each theorem and proof step
  - 38 theorems without proof steps
  - 21 theorems with proof steps
  - 208 proof steps
- We can try to prove these with Natty, or send them to external provers

Prover performance (time limit: 5 seconds)

Theorems

	Natty	E	Vampire	Zipperposition
proved (of 59)	20	36	18	26
average time	0.5	0.07	0.7	0.77
PAR-2 score	6.78	3.94	7.16	5.93

Proof steps

	Natty	E	Vampire	Zipperposition
proved (of 208)	150	191	166	156
average time	0.34	0.14	0.18	0.38
PAR-2 score	3.03	0.94	2.17	2.78

How does Natty work?

1. **Translate input to a series of logical formulas**
2. Formally verify each formula

Foundations of various provers

- first-order set theory: Mizar, Metamath
- higher-order set theory: Naproche/ZF, Megalodon
- classical higher-order logic: Isabelle, HOL, **Natty**
- dependent type theory: Lean, Coq

Higher-order logic

- Terms look like typed lambda calculus
- Can express higher-order concepts
  - Peano induction is a single formula, not a schema
- Strong typing
  - no “false theorems” such as 0 = ∅
  - static checking
- Complete proof calculus (Bentkamp et al, 2023)
- Now supported by automatic provers (e.g. E, Vampire)
- Standard interchange format (THF = Typed Higher-order Format)

Translation to logic: parsing

- (Mostly) context-free grammar
  - Less than 200 lines of EBNF
- Includes typical phrases: “we deduce that”, “we see that”, ...
- Implementation using parser combinators
- About 430 lines of OCaml code

Translation to logic: proof structure

- Natty infers block structure of each proof
- Must be correct, otherwise generated formulas will be invalid
- Need to infer scope of each quantifier, assumption
- In ordinary mathematical writing, assumptions are discharged implicitly!

Proof structure: example

Theorem 8.1. Let a, b, c ℕ. a < b if and only if s(a) < s(b).

Proof. Let a, b ∈ ℕ. Suppose that a < b. Then there is some c ∈ ℕ such that a + c = b. So a + 1 + c = b + 1. Then s(a) + c = s(b), so s(a) < s(b). Now suppose that s(a) < s(b). Then there is some c ∈ ℕ such that s(a) + c = s(b). So a + 1 + c = b + 1. Then a + c = b, so a < b.

```
let a, b : ℕ
  assume a < b
    is_some c : ℕ : a + c = b
      assert (a + 1) + c = b + 1
      assert s(a) + c = s(b)
    assert s(a) < s(b)
  assume s(a) < s(b)
    is_some c : ℕ : s(a) + c = s(b)
      assert (a + 1) + c = b + 1
      assert a + c = b
    assert a < b
assert ∀a:ℕ.∀b:ℕ.(a < b ↔ s(a) < s(b))
```

Proof structure heuristics

- Broadly speaking:
  - scope of each introduced variable V ends at the last reference to V
  - an assumption remains open until either
    - its containing scope ends
    - we see a keyword such as "Now" or "Conversely"
- Detailed rules in workshop paper

Translation to logic: outputting formulas

```
let a, b : ℕ
  assume a < b
    is_some c : ℕ : a + c = b
      assert (a + 1) + c = b + 1
      assert s(a) + c = s(b)
    assert s(a) < s(b)
  assume s(a) < s(b)
    is_some c : ℕ : s(a) + c = s(b)
      assert (a + 1) + c = b + 1
      assert a + c = b
    assert a < b
assert ∀a:ℕ.∀b:ℕ.(a < b ↔ s(a) < s(b))
```

```
1. ∀a:N.∀b:N.(a < b → ∃c:N.a + c = b)
2. ∀a:N.∀b:N.(a < b → ∀c:N.(a + c = b →
           (a + 1) + c = b + 1))
3. ∀a:N.∀b:N.(a < b → ∀c:N.(a + c = b →
           (a + 1) + c = b + 1 → s(a) + c = s(b)))
4. ∀a:N.∀b:N.(a < b →
           ∃c:N.s(a) + c = s(b) → s(a) < s(b))
...
```

Assumptions in generated formulas

Suppose that $x > 10$. Also suppose that $y > 20$.
Then $x + 1 > 11$, and $y + 2 > 22$. So $(x + 1) + (y + 2) > 33$.

- Approach 1: each output formula contains active assumptions
  - φ_1: $x > 10 \land y > 20 \rightarrow x + 1 > 11$
  - φ_2: $x > 10 \land y > 20 \rightarrow y + 2 > 22$
  - φ_3: $x > 10 \land y > 20 \rightarrow (x + 1) + (y + 2) > 33$
- Approach 2: also contain results of previous steps
  - φ_1: $x > 10 \land y > 20 \rightarrow x + 1 > 11$
  - φ_2: $x > 10 \land y > 20 \land x + 1 > 11 \rightarrow y + 2 > 22$
  - φ_3: $x > 10 \land y > 20 \land x + 1 > 11 \land y + 2 > 22 \rightarrow (x + 1) + (y + 2) > 33$
- Natty uses the second approach
  - Advantage: each output formula can be proved independently
  - Disadvantage: formulas can become large

How does Natty work?

1. Translate input to a series of logical formulas
2. **Formally verify each formula**

Internal superposition-based prover

Why write a new automatic prover?

- Other provers cannot prove all proof steps quickly, or at all
- We want to be able to say that a proof step should use a certain lemma/theorem
- Other provers don’t support all THF features
  - polymorphism
  - tuples
- More flexible / easy to integrate

Natty’s internal prover

- Broadly similar to E (and probably Vampire)
- Based on higher-order superposition calculus
  - “Superposition for Higher-Order Logic” (Bentkamp et al, 2023)
- The full calculus is complete, but complex
- Natty uses a pragmatic, incomplete variant (like E)
- Goal: prove easy theorems quickly (e.g. less than 5 seconds)

Proof calculus: superposition rule

$$\frac{\overbrace{D' \lor t \approx t'}^{D} \quad C\langle u \rangle}{(D' \lor C\langle t' \rangle)\sigma}\ \textsc{Sup} \qquad \sigma \in \mathrm{csu}(t, u)$$

(i) u is not fluid
(ii) u is not a variable
(iii) $t\sigma \not\precsim t'\sigma$
(iv) the position of u is eligible in C w.r.t. σ (*)
(v) $C\sigma \not\precsim D\sigma$
(vi) $t \approx t'$ is maximal in D w.r.t. σ
(vii) $t\sigma$ is not a fully applied Boolean logical symbol
(viii) if $t'\sigma = \bot$, u is at the top level of a positive literal

Proof calculus: other rules

- Equality resolution
- Outer clausification
- Splitting clausification
- Rewriting
- Subsumption
- Simplification
- Tautology deletion
- AC (associative-commutative) tautology deletion
- Most are similar to rules in E

Proof procedure: term ordering

- Higher-order superposition calculus has technical requirements on ordering
- Natty uses suggested term ordering
  - encode higher-order terms as first-order terms
  - transfinite Knuth-Bendix ordering on first-order terms
  - allows symbols to have infinite weights
- Unary function symbols have weight 2, others have weight 1
- May still experiment with lexicographic path ordering

Proof procedure: unification

- Full higher-order unification is needed for completeness
- But it’s hard
  - only semi-decidable
  - two terms may have an infinite number of unifiers
- Natty performs only first-order unification, mostly
- Can also unify lambda terms with variables in same positions
  - e.g. $\lambda x.f(x, y)$ and $\lambda z.f(z, 4)$

Proof procedure: unification

- Natty's simple unification can still find inductive proofs
- Peano axiom of induction
  - $\forall P:(\mathbb{N} \to \mathbb{B}).(P(0) \to \forall k:\mathbb{N}.(P(k) \to P(s(k))) \to \forall n:\mathbb{N}.P(n))$
- Final consequent is $\forall(\lambda n:\mathbb{N} . P(n))$
  - which η-reduces to $\forall(P)$
- Suppose we are proving $\forall a:\mathbb{N} . 0 + a = a$
- This is $\forall(\lambda a:\mathbb{N} . 0 + a = a)$
  - which unifies trivially with $\forall(P)$
  - No higher-order unification is necessary!
- However, we must relax one superposition condition to allow this to proceed

Proof procedure

- Modeled after main loop in E
- Input: formula to be proved, plus all known formulas
- Negate the goal, then saturate to search for a contradiction

Proof procedure: main loop

- Natty uses DISCOUNT loop as found in E
- Clauses are in two sets: processed = *P* and unprocessed = *U*
- Loop:
  1. Select a **given clause** *C* from *U*, add it to *P*
  2. Simplify *C* using clauses from *P*
  3. Simplify clauses in *P* using *C*
  4. Generate new clauses from *P* and *C*
  5. Send new and simplified clauses to *U*
- Invariant: all clauses in *P* are always mutually reduced

A surprisingly challenging proof step

Cancellation of multiplication

Theorem 5. Let a, b, c ∈ ℕ. If c ≠ 0 and ac = bc then a = b.

Proof. Let

G = { x ∈ ℕ | for all y, z ∈ ℕ, if z ≠ 0 and xz = yz then x = y }.

Let b, c ∈ ℕ with c ≠ 0 and 0 · c = bc. Then bc = 0. Since c ≠ 0, we must have b = 0 by Theorem 4.1. So 0 = b, and hence 0 ∈ G.

Now let a ∈ ℕ, and suppose that a ∈ G. Let b, c ∈ ℕ, and suppose that c ≠ 0 and s(a) · c = bc. Then by Theorem 3.5 we deduce that ca + c = bc. If b = 0, then either s(a) = 0 or c = 0, which is a contradiction. Hence b ≠ 0. By Lemma 1 there is some p ∈ ℕ such that b = s(p). Therefore ca + c = s(p) · c, and we see that ca + c = cp + c. It follows by Theorem 2.1 that ca = cp, so ac = pc. By hypothesis it follows that a = p. Therefore s(a) = s(p) = b. Hence s(a) ∈ G, and we deduce that x ∈ G for all x ∈ ℕ.

- This proof step should be trivial, but none of E, Vampire, Zipperposition can prove it in 5 seconds!

Proof procedure: pinning

```
∀b:ℕ.∀c:ℕ.(c ≠ 0 → 0 · c = bc → 0 = b)
→ ∀y:ℕ.∀z:ℕ.(z ≠ 0 → 0 · z = yz → 0 = y)
→ ∀a:ℕ.(∀y:ℕ.∀z:ℕ.(z ≠ 0 → az = yz → a = y)
        → ∀b:ℕ.∀c:ℕ.(c ≠ 0
                        → s(a) · c = bc
                        → ca + c = bc
                        → (b = 0 → ⊥)
                        → b ≠ 0
                        → ∀p:ℕ.(b = s(p)
                                → ca + c = s(p) · c
                                → ca + c = cp + c
                                → ca = cp)))
```

- `ca + c = cp + c` gets rewritten, so it can't unify with the antecedent of a relevant theorem
- Natty **pins** clauses derived from the goal, so it can prove this step

Proof procedure: given clause selection

- Critical for prover performance
- Most superposition provers use two or more priority queues
  - e.g. one queue ordered by age, one queue by term size
  - select in round robin fashion
- Natty uses a single queue with a single cost function
- Intution: in many proofs most steps are downhill
- A clause’s cost is the number of uphill steps in its derivation

Proof procedure: given clause selection

- Every superposition inference has a cost δ
  - All other inferences (e.g. rewriting) have cost 0
- Let $w(C)$ = Knuth-Bendix weight of clause C
- Suppose that E is derived from D, C by superposition
  - If $w(E) \leq w(C)$ (i.e. a downhill step), then $\delta = 0.01$
  - Otherwise $\delta = 1.0$
- The cost k of each clause is the total cost of all inferences in its derivation
- Natty finds all these inferences via a depth-first search
- A clause’s cost is not the sum of the costs of its parents!

Advantages of a single cost function

- Easier to understand / debug
- We can encourage the prover to use certain axioms / known theorems by decreasing their initial cost (e.g. to a negative value)
  - Not yet implemented

Proof procedure: clausification

- Clause normal form in first-order logic
  - clause = $L_1 \lor \ldots \lor L_n$
  - Each L_i is a literal $P(t_1, \ldots, t_n)$ or $\neg P(t_1, \ldots, t_n)$
  - All variables implicitly universally quantified
- Any first-order formula can be transformed to a conjunction of clauses
- Satisfiability is preserved
- Existential quantifiers eliminated via Skolemization

Proof procedure: clausification

- Some higher-order provers (E) clausify all formulas immediately
- Higher-order inferences can generate formulas with quantifiers
  - E will immediately clausify those as well
- Clausification destroys formula structure
  - Makes proofs hard to understand
  - Formula structure can be useful for inferences

Proof procedure: clausification

- Natty tries to preserve formula structure as much as possible
- No immediate clausification
- However, formulas must be clausified sooner or later
- Dilemma: should clausification be destructive?
  - If yes, then formula structure is lost
  - If no, then many formulas will be redundant

Two clausification rules

- Rule OC performs a clausification step that does not split the clause, e.g.
  - $\neg(A \land B)$ becomes $(\neg A \lor \neg B)$
  - $(A \rightarrow B)$ becomes $(\neg A \lor B)$
  - eliminate universal quantifier $\forall$
  - skolemize existential quantifier $\exists$
- Rule SPLIT performs a step that splits the clause into two, e.g.
  - $\neg(A \lor B)$ becomes clauses $\neg A$ and $\neg B$
  - $\neg(A \rightarrow B)$ becomes clauses A and $\neg B$

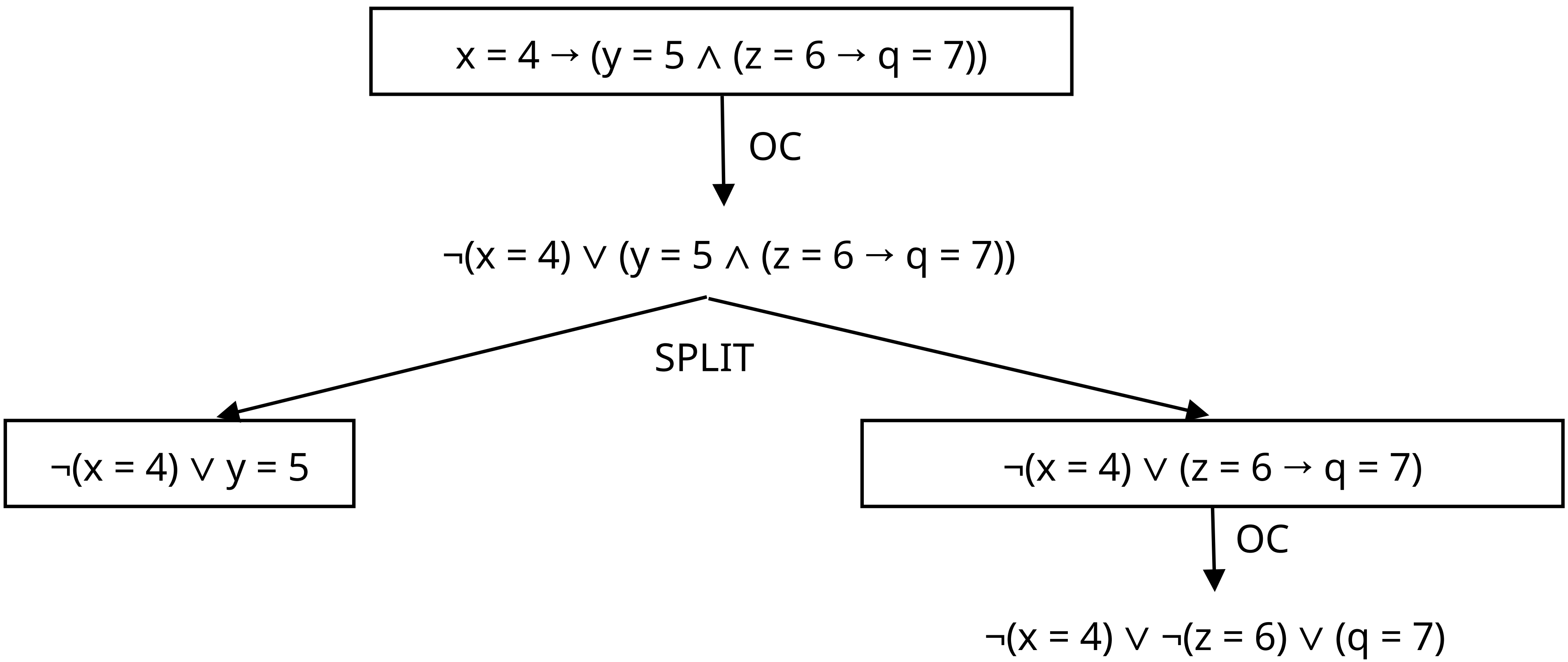
Dynamic clausification

- To perform superposition between clauses C and D:
- Apply OC repeatedly to C: $C_1 , ... , C_n$
- Apply OC repeatedly to D: $D_1 , ... , D_n$
- Look for superposition inferences between pairs C_i /D_j
- Only consider literals that first appeared in C_i
- Only consider subterms that first appeared in D_j
- $C_1 , ... , C_n$ and $D_1 , ... , D_n$ are then discarded

New clause processing

- When a new clause is given:
  - Natty applies OC and SPLIT recursively to reduce it to normal form
  - Only the original clause plus immediate children of SPLITs are kept

Dynamic clausification: example



Next steps: improve prover performance

- Goal: Prove all steps in all theorems about $\mathbb{N}$ and $\mathbb{Z}$
- Experiment with given clause heuristic
- Index clauses
- Profile to find bottlenecks

Next steps

- Goal: prove first 10 Wiedjik theorems
- Add type polymorphism, possibly with type inference
- Allow inductive type definitions
- Allow recursive function definitions
- Allow new type definitions
- Define reals and rational numbers

Questions