A Natural-Language Proof Assistant for Higher-Order Logic (Work in Progress)

Adam Dingle[1]

[1]KSVI, Faculty of Mathematics and Physics, Charles University, Prague

Abstract

Natty is a new proof assistant in an early stage of development. It can read input written in a controlled natural language that looks like mathematical English with a restricted grammar and vocabulary. It translates this input to a series of formulas of classical higher-order logic. It can export these formulas to files in the standard THF format, or can attempt to prove them itself using a built-in automatic prover based on the higher-order superposition calculus. The built-in prover clausifies formulas incrementally to preserve as much structure as possible as it performs inferences. Although Natty is small (less than 2500 lines of OCaml code), its performance seems to be competitive with established provers such as E and Vampire in proving some basic arithmetic identities involving induction over natural numbers. In addition, the THF files that it generates for arithmetic identities may serve as a useful test set for other higher-order provers.

Keywords

controlled natural language, formalization, theorem proving, higher-order logic, superposition

1. Introduction

Natty is a new natural-language proof assistant with an embedded automatic prover based on higher-order superposition. An input file for Natty is written in a controlled natural language [1, 2] that looks like mathematical English with a restricted grammar and vocabulary. It may contain a series of axioms, definitions, and theorems, which may optionally include proofs written in natural language. Natty translates this input to a series of formulas of classical higher-order logic, each representing an entire theorem or a proof step. Natty then attempts to prove these formulas using its embedded prover, reporting success or failure to the user.

Natty is an early work in progress, and its capabilities are limited at this time. It is at least able to read a natural-language input file that defines the natural numbers using the Peano axioms and asserts a series of basic theorems about the naturals, such as that addition and multiplication are associative and commutative. Natty's embedded prover can prove most of these theorems in less than 1 second per theorem on a typical machine. Many of these proofs require a higher-order induction step over the natural numbers, which Natty can infer automatically. Additionally, some theorems in this input file contain hand-written proofs. Natty can translate each step in these proofs to a separate formula, and can verify almost all of these steps automatically.

Natty can also export theorems and proof steps to files in the standard THF (Typed Higher-order Form) format [3]. This is useful for comparing performance between Natty and other provers, and in fact THF files with textbook theorems about natural numbers may form an interesting test set in their own right.

Today, Natty cannot do much more than this. However the program is under active development, and our goal is to evolve it into a system that can be used for verifying mathematics in arbitrary domains. In the months to come, we plan to extend its parser, logic, and prover to be adequate for defining the integers and rationals and for verifying results in elementary number theory such as the Fundamental Theorem of Arithmetic, i.e. that each positive integer has a unique prime factorization. More mathematics will follow after that.

PAAR'24: 9th Workshop on Practical Aspects of Automated Reasoning, July 2, 2024, Nancy, France

adam.dingle@mff.cuni.cz (A. Dingle)

0000-0003-2343-906X (A. Dingle)

Natty is publicly available [4] under a permissive open source license. It is a relatively small program, currently consisting of only about 2,500 lines of OCaml code.

Any discussion of Natty will inevitably draw comparisons with Naproche [5], another natural-language proof assistant that has been under development for a number of years. Broadly speaking our goals are similar to those of Naproche: we want to build a proof assistant in which the user writes theorems and proofs in controlled natural language rather than interactively applying tactics as in assistants such as Isabelle [6] and Lean [7]. One difference between our systems is that Natty is based on higher-order logic, while Naproche uses first-order Morse-Kelley set theory. Another is that Natty contains an embedded prover that we hope will be able to handle all proof obligations, unlike Naproche which uses E [8] as an external prover.

2. Natural-language input

Natty reads source files written in the Natty language, a controlled natural language for writing mathematics. A file in this language has the extension `.n`, and may contain type declarations, constant declarations, axioms, definitions and theorems. To illustrate, here is the beginning of a file `nat.n` that defines the natural numbers axiomatically using the Peano axioms and then asserts a series of theorems about the naturals, loosely following the development in [9].

```
Axiom.  There exists a type ℕ with an element 0 : ℕ and a function s : ℕ→ℕ such that

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

Definition.  Let 1 : ℕ = s(0).

Lemma.  Let a : ℕ.  Suppose that a ≠ 0.  Then there is some b : ℕ such that a = s(b).

Axiom.  There is a binary operation + : ℕ→ℕ→ℕ such that for all n, m : ℕ,
  a. n + 0 = n.
  b. n + s(m) = s(n + m).

Theorem.  Let a, b, c : ℕ.

  1. If a + c = b + c, then a = b.
  2. (a + b) + c = a + (b + c).
  ...
```

Listing 1: Beginning of `nat.n`

The `Axiom` keyword introduces a series of type declarations, constant declarations, and/or axioms. We see that the first `Axiom` block above includes a type declaration for ℕ, constant declarations for `0` and `s`, and three axioms. The `Definition` keyword introduces a new constant and declares that it is equal to some other value. The `Theorem` and `Lemma` keywords are synonyms, and introduce one or more theorems. Each theorem in a single block may be individually numbered.

Notice that Unicode characters such as ℕ, 𝔹 and ≠ are allowed and in fact encouraged in Natty source files. Our goal in designing the Natty language is for the user to be able to write in a way that resembles textbook mathematics as closely as possible, within the limitations of plain text and the Unicode character set.

Also notice that each constant or variable in the source file above has a type such as ℕ, ℕ → ℕ, or ℕ → ℕ → ℕ (a curried function type). These reflect the fact that Natty is based on strongly typed higher-order logic. The type 𝔹, visible in axiom (c) above, is predefined in Natty and represents the booleans.

At the moment, the user must always specify a type when declaring a constant or variable in a Natty source file. In the future we intend to implement at least some level of type inference so that types may

sometimes be omitted.

We also see that in the file above addition is introduced axiomatically, rather than by definition in terms of lower-level concepts. In the current system, axioms such as these are the only practical method for introducing new functions. This is unfortunate, since an erroneous axiom may make the entire input inconsistent. In the future, we intend to implement a mechanism that lets the user declare inductive types and define functions recursively over those types. Then the user will not need to introduce new axioms for each new type or function. Both Isabelle [10] and HOL Light [11, 12] include this sort of inductive type mechanism.

Further down in `nat.n` we introduce multiplication axiomatically, then assert several related theorems:

```
Axiom.  There is a binary operation · : ℕ→ℕ→ℕ such that for all n, m : ℕ,

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

Theorem.  Let a, b, c : ℕ.

  1. a · 0 = 0 = 0 · a.
  2. a · 1 = a = 1 · a.
  3. c(a + b) = ca + cb.
  4. (ab)c = a(bc).
  5. s(a) · b = ab + b.
  ...
```

Listing 2: Multiplication

Notice above that Natty will implicitly multiply (using the · operator) variables that appear in adjacent succession, as in the expression `(ab)c`. It distinguishes this situation from function application, as in the expression `s(a)`, by using type information: `a` has type $\mathbb{N}$ but `s` has type $\mathbb{N} \to \mathbb{N}$. This implicit multiplication syntax is convenient, and we are not aware of any other proof assistant that offers it.

Further down in `nat.n` we see that several theorems are accompanied by hand-written proofs. Listing 3 shows one such theorem, which asserts that multiplication of natural numbers can be cancelled on the right. This theorem is too difficult for Natty (at this time) to prove automatically in its entirety, so the hand-written proof allows Natty to verify the theorem by checking one proof step at a time.

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

Proof. Let

    G : ℕ→𝔹 = { 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.
  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 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 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.
```

Listing 3: Proof of right cancellation of multiplication

At the top of this proof we see an example of Natty's set comprehension syntax. In Natty a set is identified with a function whose codomain is $\mathbb{B}$, the booleans. This is a common set representation in higher-order logic. So in fact the set comprehension $\{x : \tau \mid \phi\}$ denotes exactly the higher-order term $\lambda x : \tau \,.\, \phi$, where ϕ has type $\mathbb{B}$, though the casual user should not need to think about this much.

Notice that in the proof some steps include a justification such as `By Lemma 1` or `By hypothesis`. The current implementation of Natty ignores such justifications. However, in the future we intend to

use them as strong hints to the automatic prover about which assumptions it should use in verifying a proof step.

The expected workflow of a user using Natty is as follows. The user can create a source file and enter axioms and theorems, then run Natty, which will attempt to prove all theorems automatically. If the proof of any theorem fails, the user must edit the source file to provide a proof, and then on the next run Natty will attempt to prove each proof step in turn. If it cannot prove any step, the user must edit the source file again and make the step smaller. In the near future we plan to build an interactive environment for Natty (probably as an extension for Visual Studio Code) that will make this process easier. The Naproche component in the Isabelle/jEdit IDE [13] serves a similar purpose.

3. Logic

Natty is founded on classical higher-order logic with rank-1 polymorphism, boolean extensionality, functional extensionality, and choice, with Henkin semantics. This is essentially the same logic used in proof assistants such as Isabelle/HOL [6] and HOL Light [12], by automatic higher-order provers such as recent versions of E [14], Vampire [15], and Zipperposition [16], and by the THF specification [3]. We will not describe this logic here. A standard presentation (though without rank-1 polymorphism) is given by Andrews [17].

In fact, the current implementation of Natty does not use the full power of this logic. In the current system the equality operator $\approx$ and the quantifiers $\forall$ and $\exists$ have polymorphic types, but user-defined functions may not be polymorphic. Natty's development of the natural numbers in `nat.n` does not require extensionality or choice. Finally, as we discuss in our section on Natty's proof procedure below, Natty can make few higher-order inferences at this time.

Higher-order logic is nevertheless useful as a foundation even in this early stage of development. For example, it lets us express the Peano induction axiom in a natural way, without using an axiom schema as would be needed in first-order logic. As we have already seen, we can express sets as functions with codomain $\mathbb{B}$, which is an elegant formulation that does not require any set theory axioms. The strong type system allows Natty to perform useful type checking on the user's behalf, and may even make inference more efficient: for example, Natty's unifier will not even attempt to unify two terms with distinct types.

In Natty's logic the usual constants $\top$, $\bot$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$, and $\approx$ are predefined. Of these, only $\approx$ is accessible via a similar symbol (i.e. the ordinary equals sign =) in Natty source files. To express formulas involving the other logical constants, the user must use natural-language equivalents, e.g. `a = 3 or b = 4` for the formula $a \approx 3 \vee b \approx 4$.

In higher-order logic terms and formulas are identified, and we will generally use these words as synonyms. We use the syntax $\phi[\psi/x]$ to denote the result of substituting ψ for the variable x in ϕ.

4. Translation into logic

In this section we will describe how Natty translates an input file to a series of formulas of higher-order logic.

4.1. Parsing and type checking

Natty first parses the input file, following a mostly context-free grammar for its controlled natural language. It implements the parser using the parser combinator library MParser [18], which is patterned after Haskell's Parsec library [19]. We have found parser combinators to be a convenient and powerful tool for implementing this sort of natural-language parser.

We will not present Natty's grammar here. A summary of it can be found in the file `grammar.ebnf` in the Natty source distribution [4]. The grammar is likely to evolve quickly as we enhance Natty to handle more kinds of mathematical statements.

Natty's parser outputs a list of *statements*, each of which is a type declaration, a constant declaration, an axiom, a definition, or a theorem. A theorem that includes a natural-language proof will contain parsed *proof steps*. Each proof step is one of the following.

`Assert`(*formula*) : A step such as "`And so x > 4`", asserting that a formula is true.

`Let`(*var list*, *type*) : A step such as "`Let x, y: ℕ`", introducing new arbitrary values of a certain type.

`LetVal`(*var*, *type*, *formula*) : A step such as "`Let x = 2 + 2`", defining a local constant with a certain value.

`Assume`(*formula*) : A step such as "`Suppose that f(y) = 0`", introducing an assumption for some part of the proof.

`IsSome`(*var*, *type*, *formula*) : A step such as "`there is some p : ℕ such that b = s(p)`", introducing a value that is asserted to exist.

As a next step, Natty type checks all statements. It checks that every constant symbol appearing in an axiom, definition, or theorem has been declared, and that all function applications apply a function to a value of appropriate type. A theorem's proof steps are not type checked in this phase; that will happen later, after a proof structure is inferred. Type checking is fairly easy, since Natty does not yet perform any type inference and does not yet allow any form of polymorphism for variables or constants defined by the user.

At this point Natty's translation process is essentially complete for axioms, definitions, and theorems without proofs. However Natty must now translate every natural-language proof into a series of formulas to be proven, which is the subject of the following sections.

4.2. Inferring proof structure

After Natty has produced a series of proof steps, it will arrange them into a tree representing the *block structure* of the proof. This tree will indicate the scope of each variable and assumption in the proof. In the tree the top-level node represents the entire proof, and every other node is a proof step. Every node has a sequence of zero or more children. `Assert` steps never have children; other types of steps usually do. A preorder traversal of the tree will always produce all proof steps in their original order.

Before we present the algorithm for inferring block structure, it will be helpful to see an example of a tree that it may produce. Recall the proof of cancellation of multiplication that we saw in Listing 3 above. When run with the `-d` (debug) option, Natty will print out the structure that it infers for this proof:

```
let_val G : (ℕ→𝔹) = λx:ℕ.∀y:ℕ.∀z:ℕ.(z ≠ 0 → xz = yz → x = y)
  let b, c : ℕ
    assume c ≠ 0 ∧ 0 · c = bc
      assert bc = 0
      assert c ≠ 0
      assert b = 0
      assert 0 = b
  assert G(0)
  let a : ℕ
    assume G(a)
      let b, c : ℕ
        assume c ≠ 0 ∧ s(a) · c = bc
          assert ca + c = bc
          assert b = 0 → (s(a) = 0 ∨ c = 0) ∧ (s(a) = 0 ∨ c = 0 →⊥)
          assert b ≠ 0
          is_some p : ℕ : b = s(p)
            assert ca + c = s(p) · c
            assert ca + c = cp + c
```

```
            assert ca = cp
            assert ac = pc
            assert a = p
            assert s(a) = s(p) ∧ s(p) = b
      assert G(s(a))
  assert ∀x:ℕ.G(x)
assert ∀a:ℕ.∀b:ℕ.∀c:ℕ.(c ≠ 0 → ac = bc → a = b)
```

Listing 4: Inferred proof structure

This tree illustrates all of the proof step types that Natty uses. The root node represents the entire proof and is not shown. It has two children: the `let_val` step at the top, and the final step which asserts the theorem that is being proven.

Also notice that in this tree instances of the set membership operator $\in$ appear as function application, which is logically equivalent since sets are identified with functions. For example, `0 ∈ G` appears here as `G(0)`.

The tree structure indicates where each introduced assumption will be discharged, and where each introduced variable's scope will end. In a natural-language proof, this information is usually implicit. For example, the proof in Listing 3 introduces an assumption "`suppose that c ≠ 0 and s(a) · c = bc.`" Later on, the proof asserts "`Hence s(a) ∈ G`". This assertion must not fall within the scope of the assumption, for then it would be conditional and would be too weak to participate in the final induction step. In fact this assumption must be discharged immediately before this assertion is made. In Listing 4, we see that the assertion (which appears as `G(s(a))`) is correctly outside of the scope of the assumption.

To construct the structure for a proof, Natty takes the steps parsed from the user-provided proof and appends a final step asserting the formula of the theorem that is being proven. It then infers the proof structure using a recursive algorithm that applies the following heuristics:

- A `LetVal`, `Let` or `IsSome` proof step S encloses a scope that is *as small as possible*, subject to these constraints:
  - It must extend far enough to enclose all proof steps that refer to variables that S introduces.
  - It must extend far enough to enclose the scopes of all its children.
- An `Assume` proof step's scope extends to the end of the scope of its nearest enclosing `LetVal`, `Let` or `IsSome` step.
- An `Assert` proof step never has children, so there is no need to infer a scope for it.

The effects of these heuristics are visible in Listing 4. For example, the second occurrence of the step `let b, c : ℕ` encloses a scope that extends only to the assertion `s(a) = s(p) ∧ s(p) = b`, because that is the last step that uses either of the variables `b` or `c`. That forces the scope of the assumption `c ≠ 0 ∧ s(a) · c = bc` to end as well, so the assertion `G(s(a))` correctly falls outside that assumption.

These heuristics seem promising, however we have only tested them on the three proofs in `nat.n`. We hope they will work well for future proofs. In the worst case we may find that we need to introduce some way for proof authors to specify block structure explicitly in some cases.

4.3. Generating formulas

After Natty has inferred a proof's structure, it will generate a series of formulas representing the logical deductions made in each step in the proof. Each of these formulas will be independent, in the sense that it can be proven by Natty (or another prover) without the presence of the other proof steps as given hypotheses. The conjunction of these formulas will imply the theorem statement, so that if Natty proves them all then the theorem is proven.

Each `Assert` step (and also each `IsSome` step, which is another kind of assertion) will generate a formula, and other steps will transform these formulas in various ways. Each generated formula will

capture the proof state at the moment that an assertion is made. This proof state will reflect both the steps that *enclose* the assertion in the proof structure, and also the steps that *precede* it in the proof.

Let us first consider the effect of enclosing proof steps. Broadly speaking:

- If a step `Let([`x`], `τ`)` encloses an assertion ϕ, then the assertion will be transformed to $\forall x : \tau \,.\, \phi$, because the proof is asserting that ϕ is true for any x.
- If a step `LetVal(`x`, `τ`, `ψ`)` encloses an assertion ϕ, then the assertion will be transformed to $\phi[\psi/x]$, substituting the constant value of x into the assertion.[1]
- If a step `Assume(`ψ`)` encloses an assertion ϕ, then the assertion will be transformed to $\psi \to \phi$.
- If a step `IsSome(`x`, `τ`, `ψ`)` encloses an assertion ϕ, then the assertion will be transformed to $\psi \to \phi$, because the proof is asserting that ϕ is true in the presence of an x with the given property. In addition, the `IsSome` step will generate its own formula asserting that the given x exists.

We will make these ideas more precise in the definition of generated formulas below.

Now let us consider the effect of preceding proof steps. Any step in a natural-language proof may conceivably use the results of any previous steps in the proof. In some cases a proof author may explicitly indicate a previous step that is being used, but very often this will be implicit. And so when we generate a formula representing an assertion ϕ, we could conceivably include all the results of all steps $S_1 \ldots S_n$ that occurred anywhere previously in the proof. This would yield a formula such as $S_1 \to S_2 \to \ldots \to S_n \to \phi$.

A disadvantage of such an approach is that the generated formulas could become very large for steps that occur late in a large proof. In fact, most proof steps depend only on the previous step and possibly some other recent or important steps. So we might choose to include a select subset of previous steps in each generated formula.

In fact Natty takes this approach and only includes certain previous steps in each formula. If a node's children are a sequence of assertions $\phi_1 \ldots \phi_n$ then Natty includes each of these assertions as an assumption when proving the others, generating the sequence of formulas $\phi_1, \phi_1 \to \phi_2, \phi_1 \to \phi_2 \to \phi_3, \ldots, (\phi_1 \to \ldots \to \phi_n)$.

However only the *conclusion* of this sequence, namely the formula ϕ_n, is used as a known fact in other parts of the proof. In other words, the formulas $\phi_1 \ldots \phi_{n-1}$ are treated as local, used only as helpers for proving the conclusion ϕ_n.

This approach dramatically cuts down on the size of generated formulas. It seems to capture the relevant facts needed for proving each proof step in the proofs we have verified. However, it is unclear whether it will be adequate for larger or more complex proofs. If it is not, we may need to adapt or abandon it, or provide a mechanism whereby the user can explicitly identify previous proof steps to be used in certain cases.

With these ideas in mind, we can now formally define the formulas that are generated from a proof structure tree. For each tree node, we will define the formulas *generated* by the node and also the node's *conclusion*, which is also a formula. Also, for any sequence of nodes (which will always be the children of a parent node), we will define the formulas generated by the sequence. These notions are defined via mutual recursion as follows:

- A node `Assume(`ϕ`)` generates the formula ϕ. Its conclusion is ϕ.
- A node `Let([`x`], `τ`)` generates the formulas $\forall x : \tau \,.\, \phi_1, \ldots, \forall x : \tau \,.\, \phi_n$, where $\phi_1, \ldots, \phi_n$ are generated by the sequence of its children. Its conclusion is $\forall x : \tau \,.\, \phi$, where ϕ is the conclusion of its last child.

[1] Alternatively we could transform the assertion to $\forall x : \tau \,.\, x = \psi \to \phi$. It is not clear which alternative is best; we intend to experiment with this in our implementation.

- A node `LetVal(`x`, `τ`, `ψ`)` generates the formulas $\phi_1[\psi/x], \ldots, \phi_n[\psi/x]$, where $\phi_1, \ldots, \phi_n$ are generated by the sequence of its children. Its conclusion is $\phi[\psi/x]$, where ϕ is the conclusion of its last child.
- A node `IsSome(`x`, `τ`, `ψ`)` generates the formulas $\exists x : \tau \,.\, \psi, \psi \to \phi_1, \ldots, \psi \to \phi_n$, where $\phi_1, \ldots, \phi_n$ are generated by the sequence of its children. Its conclusion is the conclusion of its last child.
- For a sequence of nodes $S_1, \ldots, S_n$, let $\phi_1, \ldots, \phi_n$ be the conclusions of each node in the sequence. For each formula ψ_{ij} generated by a node S_i, the sequence generates the formula $\phi_1 \to ... \to \phi_{i-1} \to \psi_{ij}$.
- The top-level node, representing the entire proof, generates the formulas generated by its sequence of children. Its conclusion is the conclusion of its last child (which is always the theorem being proven).

Theorem. *Given a tree of proof steps for a theorem, the conjunction of the formulas generated by the tree's root node implies that the theorem is true.*

Proof. A proof sketch is as follows. By induction on the depth of the proof step tree, we can show that the conjunction of the formulas generated by any node implies that node's conclusion, and that the conjunction of the formulas generated by any sequence implies the conclusion of the last node in the sequence. The conclusion of the root node is the theorem statement, so the result follows. □

5. Proof calculus

Natty's proof calculus is a pragmatic, incomplete variant of the higher-order superposition calculus oλSup developed recently by Bentkamp et al. [20] As such, it operates on the same clauses used in that calculus. We will review the relevant definitions of terms, literals, and clauses here.

In oλSup, a *term* is defined as a $\beta\eta$-equivalence class of λ-terms, which are in turn α-equivalence classes of raw λ-terms. We adopt these notions. In most situations Natty stores formulas in $\beta\eta$-reduced form. oλSup also defines a *$\beta\eta Q_\eta$-normal form* of λ-terms by applying a rewrite rule Q_η to ensure that the quantifiers $\mathsf{\forall}$ and $\mathsf{\exists}$ are only applied to λ-terms. We have not implemented this rule in Natty at this time.

A *literal* is an equation $s \approx t$ or disequation $s \not\approx t$ of terms. Equality is not ordered, so $s \approx t$ is the same as $t \approx s$. Literals are purely equational, so a non-equational formula ϕ must be encoded as $\phi \approx \mathsf{\top}$ and $\neg\phi$ as $\phi \approx \mathsf{\bot}$.

A *clause* is a finite multiset of literals $L_1 \vee \ldots \vee L_n$ representing a disjunction.

Starting with this section of the paper, we print logical symbols in bold (as in [20]) to distinguish them from the syntax of literals and clauses.

oλSup defines the notion of a *green subterm* of a λ-term, and many rules in oλSup operate only on green subterms. Essentially a green subterm is at a position where a first-order inference could take place. We adopt the notion of green subterms; see [20], section 3 for a definition. As in oλSup, we write $C\langle t\rangle$ to denote a clause C with a green subterm t.

Bentkamp et al. also define *blue subterms*, which are like green subterms but may also occur at some positions under quantifiers. We will sometimes use these too; see [20], section 3.7 for a definition. We write $C\langle\!\langle t\rangle\!\rangle$ to denote a clause C with a blue subterm t.

oλSup specifies that the input set of clauses must be preprocessed by two rewrite rules $\mathsf{\forall}_\approx$ and $\mathsf{\exists}_\approx$ to eliminate quantified variables in certain higher-order contexts. This is required for completeness, however we have not implemented this preprocessing in Natty at this time.

As in oλSup, Natty's calculus is parameterized by a *strict term order* $\prec$ that must be well-founded, total on ground terms, stable under grounding substitutions, and also satisfy certain other conditions listed in Definition 15 in [20]. We will not repeat these conditions here. $\preceq$ is the reflexive closure of $\prec$.

We will describe the specific term order that Natty uses below. As is common practice, the term order $\prec$ is lifted to literals and clauses via the multiset encoding described in [21], section 2.4.

As in oλSup, a *fluid term* is either a variable applied to one or more arguments, or a λ-expression t such that for some substitution σ, $t\sigma$ is no longer a λ-expression due to η-reduction. As suggested in [20], section 5, Natty assumes that any non-ground λ-expression is fluid.

oλSup also defines which positions are *eligible* in a clause with respect to a substitution σ. We will adopt this notion as well. The definition of an eligible position in oλSup is slightly complicated due to the possibility of selected literals. Natty does not support literal selection, which simplifies this notion a bit. Consider a literal L to be maximal in a clause C with respect to σ if $L\sigma$ is maximal in $C\sigma$ with respect to the term ordering $\prec$. Then essentially a green position is eligible in C with respect to σ if it can be reached by descending the term structure of a maximal literal while never passing through a term s in an equation $s \approx t$ where $s\sigma \preceq t\sigma$. See [20], section 3.3 for a precise definition.

In the rules below, $\mathrm{csu}(t, u)$ refers to the complete set of unifiers between two terms t and u.

5.1. Generating rules

Natty generates new clauses using the two following rules. A side condition marked with an asterisk (*) will be relaxed by Natty in some circumstances, as discussed in a following section.

Superposition:

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

(i) u is not fluid
(ii) u is not a variable
(iii) $t\sigma \not\preceq t'\sigma$
(iv) the position of u is eligible in C w.r.t. σ (*)
(v) $C\sigma \not\preceq D\sigma$
(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

The clauses C and D must have no free variables in common.

This is a slightly simplified form of the superposition rule from oλSup. The most notable difference is that oλSup may perform superposition into a variable u in some circumstances, but our rule never does. (This presumably breaks completeness, but our pragmatic calculus is not complete anyway.) Also, our condition (vii) only excludes fully applied Boolean logical symbols (i.e. $\top$, $\bot$, $\neg$, $\wedge$, $\vee$, $\rightarrow$), whereas the corresponding condition in oλSup excludes all logical symbols in this context. We relaxed this condition because we found that a superposition step with a fully applied quantifier ($\forall$ or $\exists$) can be an important inference in some proofs.

Equality resolution:

$$\frac{C' \vee u \not\approx u'}{C'\sigma}\ \textsc{ERes} \qquad \sigma \in \mathrm{csu}(u, u')$$

(i) $u \not\approx u'$ is maximal in C w.r.t. σ

This is a slightly simplified form of the equality resolution rule from oλSup. (Natty does not actually implement the side condition (i) at this time, but we will implement that soon.)

Natty should also include an equality factoring rule, which is necessary for first-order completeness. However, we have not implemented one yet.

5.2. Clausifying rules

Natty uses these rules to clausify formulas as described in a following section. We have written them with a double line to indicate that they could be applied destructively (i.e. discarding the parent formula) while retaining completeness. However, we will see later that Natty does not always apply these rules destructively.

Outer clausification:

$$\frac{\frac{s \approx \mathsf{T} \vee C}{}}{\mathrm{oc}(s) \vee C}\ \text{OC} \qquad \frac{\frac{s \approx \bot \vee C}{}}{\mathrm{oc}(\neg s) \vee C}\ \text{OC}$$

This rule is a variant of the OuterClaus rule in [22]. However, unlike that rule it never splits a clause into two. The function oc is defined on formulas as follows:

$$\begin{gathered}
\mathrm{oc}(s \vee t) = (s \approx \mathsf{T} \vee t \approx \mathsf{T}) \\
\mathrm{oc}(s \rightarrow t) = (s \approx \bot \vee t \approx \mathsf{T}) \\
\mathrm{oc}(s \approx t) = (s \approx t) \\
\mathrm{oc}(\forall x.s) = (s[y/x] \approx \mathsf{T}) \\
\mathrm{oc}(\exists x.s) = (s[\mathsf{k}(\bar{y})/x] \approx \mathsf{T}) \\
\mathrm{oc}(\neg(s \wedge t)) = (s \approx \bot \vee t \approx \bot) \\
\mathrm{oc}(\neg(s \approx t)) = (s \not\approx t) \\
\mathrm{oc}(\neg(\forall x.s)) = (s[\mathsf{k}(\bar{y})/x] \approx \bot) \\
\mathrm{oc}(\neg(\exists x.s)) = (s[y/x] \approx \bot)
\end{gathered}$$

In the equations above, k is a new constant, y is a variable not appearing in s or the adjacent clause C, and $\bar{y}$ represents all free variables appearing in $\exists x.s$ or $\neg(\forall x.s)$.

Splitting:

$$\frac{\frac{s \approx \mathsf{T} \vee C}{}}{\mathrm{sp}(s, C)}\ \textsc{Split} \qquad \frac{\frac{s \approx \bot \vee C}{}}{\mathrm{sp}(\neg s, C)}\ \textsc{Split}$$

This rule is derived from the same OuterClaus rule in [22], and splits clauses into two. The function sp is defined on formulas as follows:

$$\begin{gathered}
\mathrm{sp}(s \wedge t, C) = \{s \approx \mathsf{T} \vee C,\ t \approx \mathsf{T} \vee C\} \\
\mathrm{sp}(\neg(s \vee t), C) = \{s \approx \bot \vee C,\ t \approx \bot \vee C\} \\
\mathrm{sp}(\neg(s \rightarrow t), C) = \{s \approx \mathsf{T} \vee C,\ t \approx \bot \vee C\}
\end{gathered}$$

5.3. Contracting rules

Natty includes additional rules that can rewrite, subsume, and simplify clauses. It also includes rules that delete propositional tautologies and AC tautologies [8, 23] that can be proven from the associativity and commutativity axioms for operators that are known be associative and commutative. All of these rules are similar to those found in other superposition-based provers such as E [8]. For reasons of limited space, we omit a detailed presentation of these rules here.

6. Proof procedure

In this section we describe Natty's automatic prover, which finds proofs constructed from the inference rules outlined in the previous sections. Natty is broadly similar to other superposition-based provers, but its mechanisms for selecting the next given clause and for performing clausification are somewhat unusual. We will first describe some fundamental building blocks including Natty's term ordering and unification mechanism, and then move on to discussing its full proof procedure.

6.1. Term ordering

As mentioned above, the calculus oλSup requires a term ordering that satisfies certain technical conditions listed in Definition 15 in [20]. Because Natty uses the superposition rule from this calculus, it inherits these term ordering requirements. In section 3.9 of their paper, Bentkamp et al. suggest a concrete term ordering that satisfies their requirements. To compare two higher-order terms, they map each of them to a first-order term using a certain encoding, then use the transfinite Knuth-Bendix order [24] to compare the resulting first-order terms using certain weights. Natty uses this suggested term ordering and implements this mapping scheme and the Knuth-Bendix order. Instead of using the ordinal number ω for the weights of the symbols $\mathbf{\forall_1}$ and $\mathbf{\exists_1}$, Natty simply uses a large integer constant, which is effectively equivalent. Natty gives all other constant symbols a weight of 1, except unary function symbols which have a weight of 2. (We assigned this weight because it seems to facilitate certain helpful reductions in proofs about the natural numbers. However, these weight values are certainly subject to further experimentation.)

6.2. Unification

A complete higher-order prover must perform full higher-order unification, which is challenging and expensive because it is only semi-decidable, and because two terms may have an infinite number of unifiers that are not instances of one another. Natty is a pragmatic, incomplete prover and in fact its unification procedure is almost entirely first-order. Natty performs first-order unification using a simple recursive procedure including an occurs check. In addition, it can unify two lambda terms if the variables bound by the lambdas occur in exactly the same positions in those terms, and the remaining subterms of the lambda terms are unifiable. An example of such terms is the pair $\lambda x.f(x, y)$ and $\lambda z.f(z, 4)$. This is the entire extent of Natty's unification capabilities today.

Despite these limitations, Natty can actually perform useful unifications in which a variable will become bound to a higher-order term. For example, consider the Peano axiom of induction over the natural numbers:

$$\forall P : (\mathbb{N} \to \mathbb{B}).(P(0) \to \forall k : \mathbb{N}.(P(k) \to P(s(k))) \to \forall n : \mathbb{N}.P(n))$$

In higher-order logic, the final consequent $\forall n : \mathbb{N} \,.\, P(n)$ is represented as $\forall(\lambda n : \mathbb{N} \,.\, P(n))$. But this then η-reduces to simply $\forall(P)$. Now suppose that we are trying to prove the simple identity $\forall a : \mathbb{N} \,.\, 0 + a \approx a$. In higher-order logic, this is represented as $\forall(\lambda a : \mathbb{N} \,.\, 0 + a \approx a)$. This now unifies trivially with the consequent $\forall(P)$, yielding the substitution $\sigma = \{P \mapsto \lambda a : \mathbb{N} \,.\, 0 + a \approx a\}$. In Natty, superposition between the goal and this induction axiom finds this substitution and substitutes it into the axiom, which allows the proof by induction to proceed. This all works without any special higher-order unification machinery. And so Natty does not need a special process for instantiating induction axioms with abstractions from goal clauses, as is found in E [14] and Zipperposition [16].

6.2.1. Allowing inductive inferences

However, regarding the useful superposition inference we just described, a word of caution is in order. The restrictions of the superposition rule would not normally allow this superposition step to proceed. The goal is negated, so in this example we are performing superposition with $(\forall a : \mathbb{N}.0 + a \approx a) \approx \bot$ on the left, and the induction axiom on the right. If we have not clausified the induction axiom, restriction (viii) will not allow the superposition since $\forall(P)$ is not at the top level of a literal. We could clausify the induction axiom until $\forall(P)$ becomes a positive literal. The clausified axiom will have this form:

$$\neg P(0) \vee \neg\forall k : \mathbb{N}.(P(k) \to P(s(k))) \vee \forall(P)$$

But then restriction (iv) will not allow the superposition since the literal $\forall(P)$ will not be maximal after the substitution σ is applied. The substitution will ground all literals, and then according to the Knuth-Bendix ordering the literal with largest weight will be maximal, namely the second of the three

literals. One might hope that further clausification steps could cause $\forall(P)$ to become maximal (e.g. after the clause splits in two), but experiments with Natty show that this is not the case.

Now, we very much want to allow this superposition step, so Natty makes an exception. It considers a formula to be *inductive* if it has the form $\forall x : \tau \,.\, u$, where τ is a function type with codomain $\mathbb{B}$. If the right formula in a superposition is inductive, then Natty does not enforce the eligibility restriction (iv).

6.3. Main loop

The input to Natty's automatic prover is a formula to be proved, plus a set of formulas that are known. The known formulas are either axioms or are theorems that appeared earlier in the input file and have already been proved.

Like most other superposition-based provers, Natty negates the goal, then searches for a contradiction by saturating the set of input formulas. Because the proof calculus works on equational clauses, Natty initially converts each input formula to a clause of the form $\phi = \top$, or to $\phi = \bot$ for an input formula of the form $\neg\phi$. Unlike in some other provers, no clausification steps (i.e. applications of the rules OC or Split) are performed at this time.

Natty's main loop is modeled after the main loop in E [8], which is itself a variation of the DISCOUNT loop used in an earlier system of that name [25]. Like E, Natty keeps clauses in two sets containing *processed* and *unprocessed* clauses, respectively. In its main loop, it repeatedly selects a *given clause* from the unprocessed set and adds it the processed set. As it does so, it performs all possible simplifications of the given clause using clauses from the processed set. Furthermore, it back-simplifies clauses in the processed set using the given clause, and sends them back to the unprocessed set if they have changed. In this way, Natty maintains the invariant that all processed clauses are always reduced with respect to each other.

We will not provide pseudocode for Natty's main loop or discuss it more, because it is so similar to that of E. Instead, we will discuss Natty's mechanisms for given clause selection and for clausification, which are less similar to other provers.

6.4. Given clause selection

Most other superposition-based provers select given clauses in a weighted round-robin fashion from two or more priority queues. Each queue orders the available unprocessed clauses according to some characteristic function. One basic scheme used by some provers has two priority queues. One queue is a LIFO, which orders clauses by age, so that at any given moment the oldest available clause will be picked next. A second queue orders clauses by weight, which may be a simple count of the symbols in a clause or which may be a weighted sum of those symbols. In this queue the clause with the smallest weight will be picked next. Many variations of this scheme are possible. In [26], Schulz and Möhrmann present various selection heuristics and the results of experiments they performed to see which of them might work best.

Natty uses a different, experimental given clause selection mechanism. In Natty there is only a single priority queue, ordering all unprocessed clauses using a single cost function. Unlike in most systems, where clauses are selected based on their size (or weight), Natty's cost function reflects the *changes in size* that have occurred during the proof steps that have derived a clause. It is based on the intuition that many proofs involve only a small number (perhaps just one or even zero) of non-obvious or "uphill" steps that may increase the size of the formula to be proved, combined with a larger number of "easy" or "downhill" steps that reduce the formula size and bring the proof toward a close. For example, applying mathematical induction will usually increase the formula size and can be considered an uphill step, whereas applying a theorem that simplifies a formula is a downhill step that decreases the formula size. Natty's cost function is designed to penalize the uphill steps, and find proofs that go downhill most of the time.

To be more specific, Natty assigns a cost to each clause as follows. First, recall that the Knuth-Bendix ordering assigns a *weight* to each clause, defined as the sum of the weights of all of its symbols. As

we saw before, Natty gives all symbols a weight of 1, except unary function symbols which have a weight of 2. Let $w(C)$ be the total weight of clause C, computed as in the Knuth-Bendix ordering except considering the quantifier symbols $\forall$ and $\exists$ to have weight 0. (This is because e.g. prefixing a universal quantifier to a formula does not really change its essence, and we do not want to count that as an uphill step.)

Now we can describe how costs are calculated. Each input clause has cost 0. Suppose that a new clause E is derived from clauses D and C by superposition, where D and C are on the left and right in the superposition rule. We will compute two values $\delta(E)$ and $k(E)$, both of which will be stored in memory along with E. $\delta(E)$ represents E's cost *relative* to its right parent C, and $k(E)$ represents its *absolute* cost, which will be used for ranking it in the priority queue.

$\delta(E)$ is computed as follows. If $w(E) < w(C)$, then $\delta(E) = 0.01$. If $w(E) = w(C)$, then $\delta(E) = 0.02$. Otherwise, $\delta(E) = 1.0$.

$k(E)$ is computed to be $\delta(E) + i$, where i is the cost that is *inherited* from the parents D and C, and is computed as follows. Let A be a set of clauses containing D, C, and all ancestors of D or C in the graph of inferences. Then $i = \sum_{X \in A} \delta(X)$. To put it differently, i is the total cost of the proof that derived D and C. Natty computes it by performing a depth-first search to find all ancestors of D and C.

One might ask why we compare the weight of the new clause E with the weight of C, and not with D. One reason is that the superposition restriction $C\sigma \not\preceq D\sigma$ generally ensures that C will be larger, and in some cases D may be quite small, e.g. an identity such as $\forall x : \mathbb{N} \,.\, 0 + x \approx x$.

In searching for a proof, Natty only considers clauses whose costs fall under a fixed limit. By default, this limit is 1.3, meaning that Natty will only find a proof that contains at most one uphill step (i.e. a step costing 1.0). Any induction step will cost 1.0, so a proof by induction is not allowed to make any more uphill steps at all. Nevertheless, Natty can prove many identities about the natural numbers inductively within this cost limit.

Before considering superposition inferences between two clauses, Natty precomputes the cost i that will be inherited from those clauses, which is the minimum cost that any derived clause will have. If this i exceeds the fixed cost limit, Natty will not even look for possible inferences between the clauses. This optimization may make searches for proofs whose cost is near the limit significantly faster than in a search with an unbounded cost.

6.5. Clausification

If we start with a clause containing a single formula and exhaustively apply the inference rules OC and Split until no more applications are possible, we will eventually reach *clause normal form*. This process is called *clausification*. If the starting formula is first order, clause normal form will be a conjunction of disjunctions of atomic formulas. During the process of clausification, all quantifiers will be eliminated and new Skolem constants will be introduced for quantifiers that are effectively existential.

All provers must perform clausification in some way. Some provers such as E immediately clausify every formula at the beginning of the proof process. Even if a prover does this, higher-order inferences may create new formulas later on that are not in clause normal form. E will immediately clausify such formulas as soon as they appear.

Unfortunately, completely clausifying a formula will often create a large number of small clauses whose relationship to the original formula is difficult to understand. As a result, even if a refutation is found, the generated proof may be cryptic. A further disadvantage of immediate clausification is that it destroys high-level formula structure that may be useful for inferring higher-level proof steps, such as by unifying with the antecedent of a complex theorem.

Natty attempts to preserve formula structure as much as possible when generating inferences. However, the clausification rules OC and Split present a fundamental dilemma that all provers must deal with somehow. If a prover applies these rules destructively, then formula structure is lost. On the other hand, if it applies them non-destructively, it will introduce clauses that are redundant and in fact equivalent. This may lead to many duplicate inferences if the redundant clauses remain present.

Natty attempts to resolve this dilemma through a process of *dynamic clausification*, which works as follows. Suppose that it is time to compute all possible superpositions between clauses D and C on the left and right. Natty will apply the rule OC to D and C repeatedly as many times as is possible, generating two sequences of clauses $D_1, \ldots, D_m$ and $C_1, \ldots, C_n$. (Recall that OC never splits a clause.) All of these clauses are kept in memory.

Now Natty must look for superposition inferences. Every possible inference will involve some equational literal from D, and some green subterm from C. Natty will consider all possible pairs (D_i, C_j), where $1 \leq i \leq m$ and $1 \leq j \leq n$. It will look for superposition inferences between D_i and C_j, but only considering those literals that *first appeared* in D_i, i.e. were not already present in D_{i-1}. Similarly, it will only consider green subterms that first appeared in C_j. In this way, any superposition inference that is generated between a literal from D and a subterm from C will use the earliest D_i and C_j in which it is possible to make that inference, preserving as much formula structure as is possible.

Finally, when all possible inferences between D and C have been found, Natty will discard the sequences $D_1, \ldots, D_m$ and $C_1, \ldots, C_n$ from memory. They will be regenerated again when necessary. We believe this dynamic clausification process is actually fairly cheap, since most subterms between the dynamically generated formulas are shared in memory.

In this way, each clause in the generated set actually serves as a representative of all the clauses that can be generated from it using the rule OC.

When Natty adds a clause C to the processed set, it looks for a way to split the clause by applying OC zero or more times and then applying the rule Split. If this is possible, it will apply this rule *non-destructively*, creating two new clauses and retaining the original clause C. Any particular clause will only be split at most once. The newly created clauses will have a δ value of 0, so their costs will be the same as that of the parent clause. For that reason, they will soon be pulled from the priority queue and also enter the processed set, at which point they may split again. In this way, all clauses of the normal form of the original clause will be generated in a short time. There may be a fair number of such clauses, but none of them will be equivalent to any other.

This scheme seems to work reasonably well, but the non-destructive splitting may still sometimes cause an undesirable level of redundancy between active clauses. We intend to experiment more with variations of this scheme that may allow splitting to be destructive in some cases.

7. Performance

We evaluated Natty's performance versus that of E 3.0.08, Vampire 4.8, and Zipperposition 2.1 in proving theorems and proof steps exported to THF files from `nat.n`. As we saw above, this file defines the natural numbers axiomatically and then asserts various elementary theorems about them. Many of these theorems require proofs by induction. We ran E with the `--auto` option, and built Vampire from the `v4.8HO4Sledgahammer` tag (dated October 19, 2023) containing higher-order improvements not yet merged into its master branch. We ran all of these provers in a mode that used a single strategy.

`nat.n` contains 18 theorems. Of these, 3 include hand-written natural-language proofs, because they are too difficult for Natty to prove automatically. Natty generates a total of 43 proof steps from these proofs.

We ran the four provers on all the exported theorems, and then again separately on all the exported proof steps. We used a 5-second time limit for all proof attempts. This limit is much shorter than is typical in competitive evaluations of automatic theorem provers, but we choose it to reflect Natty's intended use as a prover that can be used interactively and quickly.

The results are in Table 1 and Table 2. We show individual results for theorems, but only aggregate results for proof steps. The average times in the tables only reflect theorems or proof steps that were actually proved, and are unaffected by failed proof attempts. The tables also show the PAR-2 score, which is the average time over all attempts, in which a failed attempt is assigned twice the time limit.

We think Natty's performance at this early stage is promising. It was able to prove almost as many theorems as any prover, and it proved the most proof steps of any of the provers. The comparison may

Table 1
Theorems

	conjecture	Natty	E	Vampire	Zipperposition
thm 1	$a \neq 0 \rightarrow \exists b{:}\mathbb{N}.a = s(b)$	0.03	0.00	0.00	timeout
thm 2.1	$a + c = b + c \rightarrow a = b$	0.05	0.20	timeout	0.02
thm 2.2	$(a + b) + c = a + (b + c)$	0.03	0.05	timeout	0.37
thm 2.3	$0 + a = a \wedge a = a + 0$	0.03	0.00	timeout	0.45
thm 2.4	$s(a) + b = s(a + b)$	0.03	0.01	timeout	0.58
thm 2.5	$a + b = b + a$	0.04	0.00	timeout	0.51
thm 2.6	$a + s(b) \neq 0$	0.03	0.00	0.01	0.01
thm 2.7	$a + s(b) \neq a$	0.15	0.01	0.01	0.34
thm 2.8	$a + b = 0 \rightarrow a = 0 \wedge b = 0$	0.65	0.00	0.01	0.03
thm 3.1	$a \cdot 0 = 0 \wedge 0 = 0 \cdot a$	0.07	0.00	timeout	timeout
thm 3.2	$a \cdot 1 = a \wedge a = 1 \cdot a$	0.07	0.01	timeout	timeout
thm 3.3	$c \cdot (a + b) = ca + cb$	0.21	0.03	timeout	timeout
thm 3.4	$(ab) \cdot c = a \cdot (bc)$	gave up	0.02	timeout	timeout
thm 3.5	$s(a) \cdot b = ab + b$	gave up	timeout	timeout	2.13
thm 3.6	$ab = ba$	0.13	0.01	timeout	2.03
thm 3.7	$(a + b) \cdot c = ac + bc$	0.20	0.00	timeout	0.05
thm 3.8	$ab = 0 \rightarrow a = 0 \wedge b = 0$	3.57	0.04	timeout	0.12
thm 4	$c \neq 0 \rightarrow ac = bc \rightarrow a = b$	timeout	timeout	timeout	timeout
	proved (of 18)	15	16	4	12
	average time	0.35	0.02	0.01	0.55
	PAR-2 score	1.96	1.13	7.78	3.70

Table 2
Proof steps

	Natty	E	Vampire	Zipperposition
proved (of 43)	41	40	39	40
average time	0.24	0.01	0.02	0.18
PAR-2 score	0.70	0.70	0.95	0.87

not be entirely fair, since the other provers may have been designed mostly to solve difficult problems after running for longer periods of time using a portfolio of strategies, not easy problems in just a few seconds. Still, Natty seems adequate for its intended use case. Furthermore, we could certainly optimize its performance more. For example, it is not currently performing any term indexing at all.

8. Future work

We do not have space here to list all the enhancements we would like make to Natty, but here are some of our plans. A major goal for the near future is to move beyond the natural numbers and prove theorems about the integers. To do this, we will need some sort of overloading, polymorphism, or type classes to handle e.g. the fact that addition on naturals and on the integers will have the same name and syntax but will be two different functions. Before long we will also need some way to define types inductively, or to define new types as isomorphic to equivalence classes of an existing type.

An interactive environment where the user can edit a proof and get feedback on it in real time is also a high priority.

Acknowledgments

Thanks to Martin Suda and the anonymous reviewers for their helpful comments.

This research was supported by the Grant Agency of Charles University (grant 128524).

References

[1] T. Kuhn, A survey and classification of controlled natural languages, Computational Linguistics 40 (2014) 121–170.
[2] T. Hales, An argument for controlled natural languages in mathematics, 2019. URL: https://jiggerwit.wordpress.com/wp-content/uploads/2019/06/header.pdf.
[3] G. Sutcliffe, The logic languages of the TPTP world, Logic Journal of the IGPL 31 (2023) 1153–1169.
[4] A. Dingle, Natty code repository, 2024. URL: https://github.com/medovina/natty.
[5] M. Cramer, B. Fisseni, P. Koepke, D. Kühlwein, B. Schröder, J. Veldman, The Naproche project: controlled natural language proof checking of mathematical texts, in: International Workshop on Controlled Natural Language, Springer, 2009, pp. 170–186.
[6] L. C. Paulson, Isabelle: A generic theorem prover, Springer, 1994.
[7] L. d. Moura, S. Ullrich, The Lean 4 theorem prover and programming language, in: Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28, Springer, 2021, pp. 625–635.
[8] S. Schulz, E–a brainiac theorem prover, AI Communications 15 (2002) 111–126.
[9] E. D. Bloch, The real numbers and real analysis, Springer Science & Business Media, 2011.
[10] S. Berghofer, M. Wenzel, Inductive datatypes in HOL—lessons learned in formal-logic engineering, in: International Conference on Theorem Proving in Higher Order Logics, Springer, 1999, pp. 19–36.
[11] J. Harrison, Inductive definitions: automation and application, in: International Conference on Theorem Proving in Higher Order Logics, Springer, 1995, pp. 200–213.
[12] J. Harrison, HOL light: A tutorial introduction, in: International Conference on Formal Methods in Computer-Aided Design, Springer, 1996, pp. 265–269.
[13] A. De Lon, P. Koepke, A. Lorenzen, A. Marti, M. Schütz, M. Wenzel, The Isabelle/Naproche natural language proof assistant, in: Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28, Springer International Publishing, 2021, pp. 614–624.
[14] P. Vukmirović, J. Blanchette, S. Schulz, Extending a high-performance prover to higher-order logic, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2023, pp. 111–129.
[15] A. Bhayat, G. Reger, A combinator-based superposition calculus for higher-order logic, in: International Joint Conference on Automated Reasoning, Springer, 2020, pp. 278–296.
[16] P. Vukmirovic, A. Bentkamp, J. Blanchette, S. Cruanes, V. Nummelin, S. Tourret, Making higher-order superposition work., in: CADE, 2021, pp. 415–432.
[17] P. B. Andrews, An introduction to mathematical logic and type theory: to truth through proof, volume 27, Springer Science & Business Media, 2013.
[18] M. Mouratov, MParser, 2024. URL: https://github.com/murmour/mparser.
[19] D. Leijen, E. Meijer, Parsec: Direct style monadic parser combinators for the real world (2001).
[20] A. Bentkamp, J. Blanchette, S. Tourret, P. Vukmirović, Superposition for higher-order logic, Journal of Automated Reasoning 67 (2023).
[21] L. Bachmair, H. Ganzinger, Rewrite-based equational theorem proving with selection and simplification, Journal of Logic and Computation 4 (1994) 217–247.
[22] V. Nummelin, A. Bentkamp, S. Tourret, P. Vukmirovic, Superposition with first-class booleans and inprocessing clausification, in: CADE, 2021, pp. 378–395.
[23] J. Avenhaus, T. Hillenbrand, B. Löchner, On using ground joinable equations in equational theorem proving, Journal of Symbolic Computation 36 (2003) 217–233.
[24] M. Ludwig, U. Waldmann, An extension of the Knuth-Bendix ordering with LPO-like properties,

in: International Conference on Logic for Programming Artificial Intelligence and Reasoning, Springer, 2007, pp. 348–362.

[25] J. Denzinger, M. Kronenburg, S. Schulz, DISCOUNT-a distributed and learning equational prover, Journal of Automated Reasoning 18 (1997) 189–198.

[26] S. Schulz, M. Möhrmann, Performance of clause selection heuristics for saturation-based theorem proving, in: Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings 8, Springer, 2016, pp. 330–345.