Recovering Formal Representations from Informalized Mizar Sentences

Chad E. Brown, Josef Urban, Jiří Vyskočil

Czech Technical University in Prague

September 2017

Outline

Introduction

- Long term goal: tools to read informal mathematics and create formalized versions.
- Stepping stone: handle cases where we already have the formal versions

Introduction

- Long term goal: tools to read informal mathematics and create formalized versions.

- Stepping stone: handle cases where we already have the formal versions
  - Flyspeck: Informal document by Hales; Formalization in HOL-light.

Introduction

- Long term goal: tools to read informal mathematics and create formalized versions.

- Stepping stone: handle cases where we already have the formal versions
  - Flyspeck: Informal document by Hales; Formalization in HOL-light.
  - MML (Mizar Mathematical Library): Mizar looks similar to informal mathematics, but is essentially formal since Mizar can check it.

Informal Examples

- What is $\overline{Y}$?

Informal Examples

- What is $\overline{Y}$?
- Closure of Y?

Informal Examples

- What is $\overline{Y}$?
- Closure of Y?
- Complement of Y?

Informal Examples

- What is $\overline{Y}$?
- Closure of Y? With respect to what topology?
- Complement of Y? With respect to what superset?

Informal Examples

- What is $\overline{Y}$?
- Closure of Y? With respect to what topology?
- Complement of Y? With respect to what superset?
- Something else?

Informal Examples

Consider use of $\overline{Y}$ for $X \setminus Y$
when Y is understood as a subset of X.

Informal Examples

Consider use of $\overline{Y}$ for $X \setminus Y$
when Y is understood as a subset of X.

- Example 1: *If* $Y \subseteq X$, *then* $\overline{\overline{Y}} = Y$.

Informal Examples

Consider use of $\overline{Y}$ for $X \setminus Y$
when Y is understood as a subset of X.

- Example 1: *If* $Y \subseteq X$, *then* $\overline{\overline{Y}} = Y$.
- Example 2: *Let* $X \subseteq X'$, $Y \subseteq X$ *and* $Y' \subseteq X'$. *If* $\overline{Y} \subseteq \overline{Y'}$, *then* $Y' \subseteq Y$.

Informal Examples

Consider use of $\overline{Y}$ for $X \setminus Y$
when Y is understood as a subset of X.

- Example 1: *If* $Y \subseteq X$, *then* $\overline{\overline{Y}} = Y$.
- Example 2: *Let* $X \subseteq X'$, $Y \subseteq X$ *and* $Y' \subseteq X'$. *If* $\overline{Y} \subseteq \overline{Y'}$, *then* $Y' \subseteq Y$.
  - Here $Y \subseteq X$ and $Y \subseteq X'$.

Informal Examples

Consider use of $\overline{Y}$ for $X \setminus Y$
when Y is understood as a subset of X.

- Example 1: *If* $Y \subseteq X$, *then* $\overline{\overline{Y}} = Y$.
- Example 2: *Let* $X \subseteq X'$, $Y \subseteq X$ *and* $Y' \subseteq X'$. *If* $\overline{Y} \subseteq \overline{Y'}$, *then* $Y' \subseteq Y$.
  - Here $Y \subseteq X$ and $Y \subseteq X'$.
  - Two readings:
    - Take $\overline{Y}$ to mean $X \setminus Y$. (false)
    - Take $\overline{Y}$ to mean $X' \setminus Y$. (true)

Mizar Mathematical Library (MML)

- Roughly 60,000 theorems with proofs (inside "articles")
- Text versions that read like informal mathematics
- The text looks potentially ambiguous, but is resolved by the environment given in the header of the Mizar article.

Mizar Mathematical Library (MML)

- Roughly 60,000 theorems with proofs (inside "articles")
- Text versions that read like informal mathematics
- The text looks potentially ambiguous, but is resolved by the environment given in the header of the Mizar article.
- Annotated XML versions obtained when Mizar checks the articles, creating versions independent of the article header.

Mizar Mathematical Library (MML)

- Roughly 60,000 theorems with proofs (inside "articles")
- Text versions that read like informal mathematics
- The text looks potentially ambiguous, but is resolved by the environment given in the header of the Mizar article.
- Annotated XML versions obtained when Mizar checks the articles, creating versions independent of the article header.
- From the XML versions various representations can be derived:
  - Parse trees
  - "Pattern representations" (partial first-order terms and propositions with missing information)
  - (MPTP) "Constructor representations" (full unambiguous first-order terms and propositions)
  - MPTP has been used for first order LTB competitions; higher order version coming soon

Key Processes and the MML Data

- Probabilistic parsing of informal versions
  - Parse trees from the XML version of the MML for training; underlying text for testing

Key Processes and the MML Data

- Probabilistic parsing of informal versions
  - Parse trees from the XML version of the MML for training; underlying text for testing
- Map parse trees to quasi-terms/quasi-propositions
  - Pattern representations of the MML definitions and theorems.

Key Processes and the MML Data

- Probabilistic parsing of informal versions
  - Parse trees from the XML version of the MML for training; underlying text for testing
- Map parse trees to quasi-terms/quasi-propositions
  - Pattern representations of the MML definitions and theorems.
- Elaborate quasi-terms/quasi-propositions into actual terms/propositions
  - Constructor representations of the MML definitions and theorems.

Key Processes and the MML Data

- Probabilistic parsing of informal versions
  - Parse trees from the XML version of the MML for training; underlying text for testing
- Map parse trees to quasi-terms/quasi-propositions
  - Pattern representations of the MML definitions and theorems.
- Elaborate quasi-terms/quasi-propositions into actual terms/propositions
  - Constructor representations of the MML definitions and theorems.
- Use these terms/propositions to interact with theorem provers
  - Call theorem provers on the constructor representations of the MML theorems.

Outline

Introduction

Parsing

Mizar Typing System

Examples

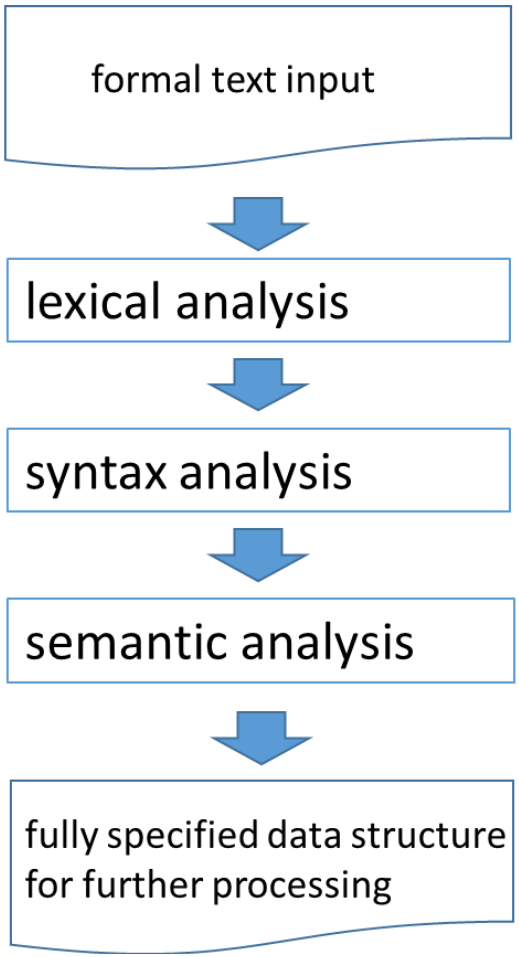
Algorithms

Results

Conclusion

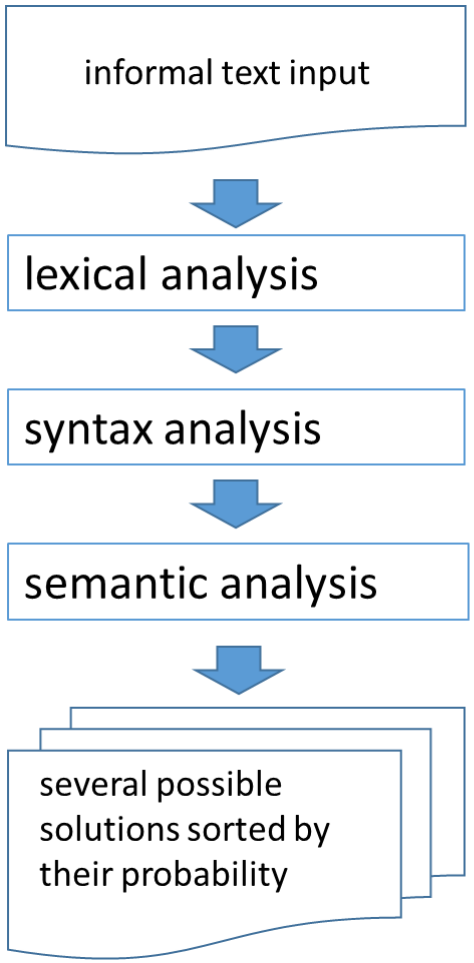
References

Traditional Parsing Approach:



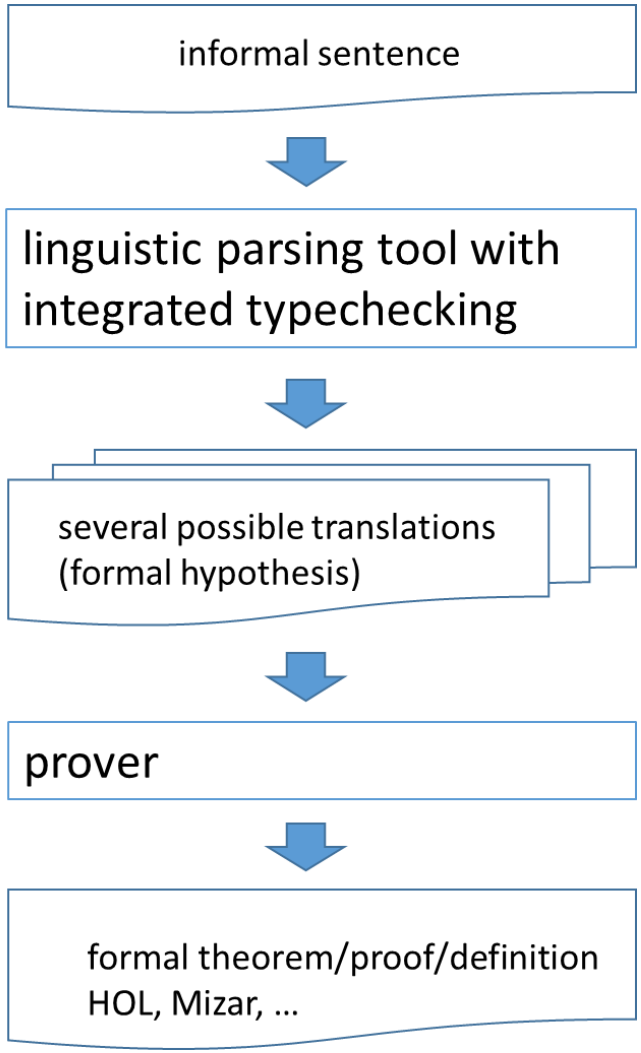
- a language is designed manually in such a way that:
  - lexical tokens can be fully specified by some *regular language*
  - syntax analyzer can be fully specified by some *unambiguous context free grammar* (typically by deterministic CFG)
  - semantic analyzer typically resolves types of symbols and subtrees in a parsing tree, checks semantic correctness of binders, ...

Linguistic Parsing Approach:

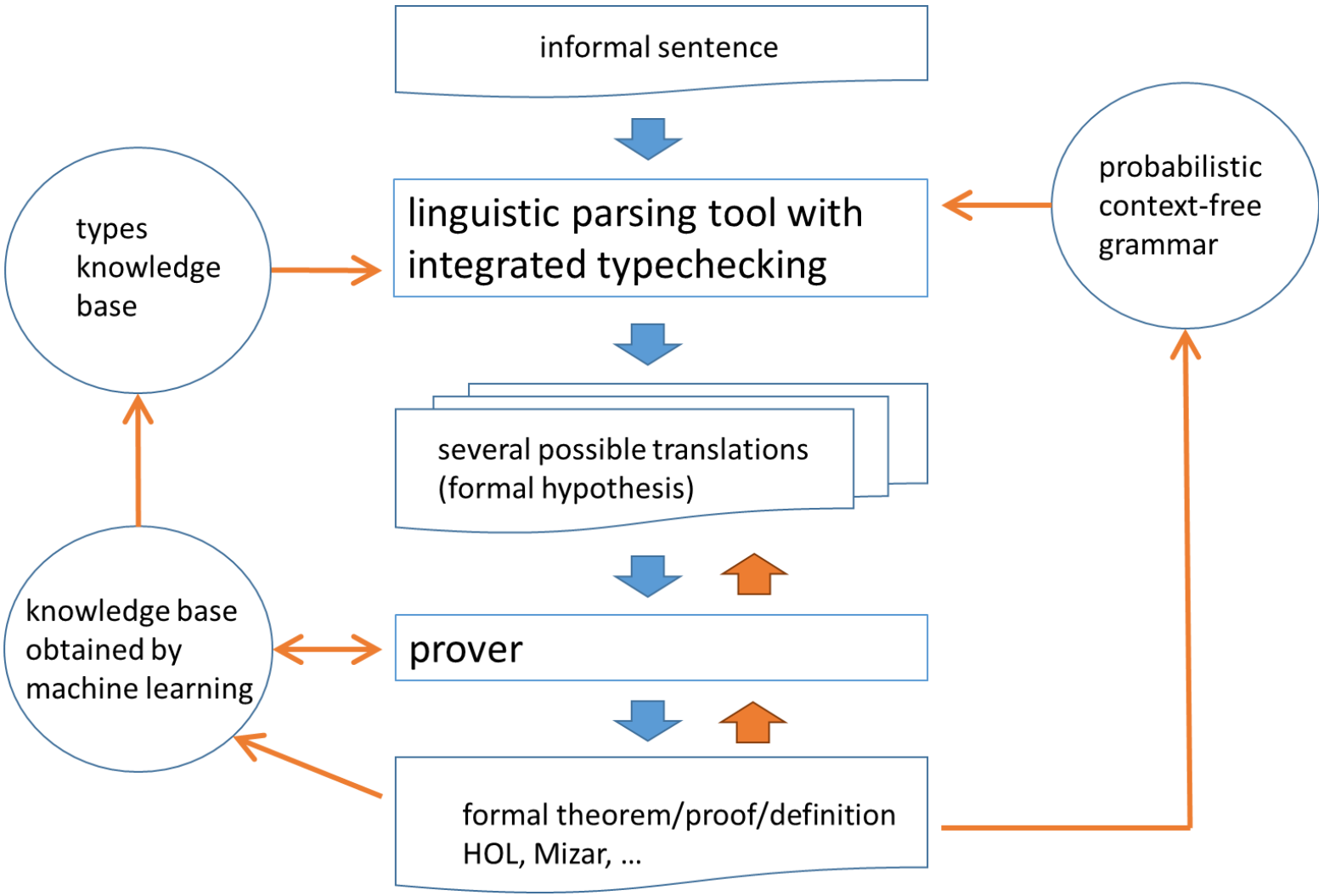


- all of these phases (or at least some of them) can be learned (instead of encoding them manually) from examples by machine learning
- syntax (and mostly even semantics) analysis can be done by ambiguous CFG with probabilities (PCFG) and lexical analysis (in case of English) is usually simple
- examples for learning have same (or similar) structure as parsing trees and they are called treebanks in this domain.
- rules and probabilities can be learned from treebanks
- CYK or Earley parser can be used for parsing such PCFG

Toolchain Overview:



Toolchain Overview:



Outline

Mizar Typing System

- Mizar uses some predicates as types.
- "natural" or "non empty" or "Element of X"
- Saying t satisfies a type means a predicate like $P(t, s_1, \ldots, s_m)$ or $\neg P(t, s_1, \ldots, s_m)$ holds.
- The typing rules can essentially be classified as follows:
  - Types of functions f: If the arguments $x_1, \ldots, x_n$ satisfy certain types, then $f(x_1, \ldots, x_n)$ satisfies certain types.

Mizar Typing System

- Mizar uses some predicates as types.
- "natural" or "non empty" or "Element of X"
- Saying t satisfies a type means a predicate like $P(t, s_1, \ldots, s_m)$ or $\neg P(t, s_1, \ldots, s_m)$ holds.
- The typing rules can essentially be classified as follows:
  - Types of functions f: If the arguments $x_1, \ldots, x_n$ satisfy certain types, then $f(x_1, \ldots, x_n)$ satisfies certain types.
  - Hierarchy rules: If x satisfies certain types, then it also satisfies other types.

Mizar Typing System

- Mizar uses some predicates as types.
- "natural" or "non empty" or "Element of X"
- Saying t satisfies a type means a predicate like $P(t, s_1, \ldots, s_m)$ or $\neg P(t, s_1, \ldots, s_m)$ holds.
- The typing rules can essentially be classified as follows:
  - Types of functions f: If the arguments $x_1, \ldots, x_n$ satisfy certain types, then $f(x_1, \ldots, x_n)$ satisfies certain types.
  - Hierarchy rules: If x satisfies certain types, then it also satisfies other types.
  - Cluster rules: A term t satisfies a certain type if its free variables satisfy certain types.

Mizar Typing System

- Mizar uses some predicates as types.
- "natural" or "non empty" or "Element of X"
- Saying t satisfies a type means a predicate like $P(t, s_1, \ldots, s_m)$ or $\neg P(t, s_1, \ldots, s_m)$ holds.
- The typing rules can essentially be classified as follows:
  - Types of functions f: If the arguments $x_1, \ldots, x_n$ satisfy certain types, then $f(x_1, \ldots, x_n)$ satisfies certain types.
  - Hierarchy rules: If x satisfies certain types, then it also satisfies other types.
  - Cluster rules: A term t satisfies a certain type if its free variables satisfy certain types.
  - Redefinitions: A given function (or relation) applied to distinct variables satisfying certain types is equal (or equivalent to) a certain term (or proposition).

Mizar Typing System

- Mizar uses some predicates as types.
- "natural" or "non empty" or "Element of X"
- Saying t satisfies a type means a predicate like $P(t, s_1, \ldots, s_m)$ or $\neg P(t, s_1, \ldots, s_m)$ holds.
- The typing rules can essentially be classified as follows:
  - Types of functions f:
  - Hierarchy rules:
  - Cluster rules:
  - Redefinitions:

Mizar Typing System

- Mizar uses some predicates as types.
- "natural" or "non empty" or "Element of X"
- Saying t satisfies a type means a predicate like $P(t, s_1, \ldots, s_m)$ or $\neg P(t, s_1, \ldots, s_m)$ holds.
- The typing rules can essentially be classified as follows:
  - Types of functions f:

    If x and y are reals, then $x + y$ is a real.
  - Hierarchy rules:
  - Cluster rules:
  - Redefinitions:

Mizar Typing System

- Mizar uses some predicates as types.
- "natural" or "non empty" or "Element of X"
- Saying t satisfies a type means a predicate like $P(t, s_1, \ldots, s_m)$ or $\neg P(t, s_1, \ldots, s_m)$ holds.
- The typing rules can essentially be classified as follows:
  - Types of functions f:

    If x and y are reals, then $x + y$ is a real.
  - Hierarchy rules:

    If x is a natural, then x is a real.
  - Cluster rules:
  - Redefinitions:

Mizar Typing System

- Mizar uses some predicates as types.
- "natural" or "non empty" or "Element of X"
- Saying t satisfies a type means a predicate like $P(t, s_1, \ldots, s_m)$ or $\neg P(t, s_1, \ldots, s_m)$ holds.
- The typing rules can essentially be classified as follows:
  - Types of functions f:

    If x and y are reals, then $x + y$ is a real.
  - Hierarchy rules:

    If x is a natural, then x is a real.

    If A and B are sets and f is a function from A to B, then f is a functional relation.
  - Cluster rules:
  - Redefinitions:

Mizar Typing System

- Mizar uses some predicates as types.
- "natural" or "non empty" or "Element of X"
- Saying t satisfies a type means a predicate like $P(t, s_1, \ldots, s_m)$ or $\neg P(t, s_1, \ldots, s_m)$ holds.
- The typing rules can essentially be classified as follows:
  - Types of functions f:

    If x and y are reals, then $x + y$ is a real.
  - Hierarchy rules:

    If x is a natural, then x is a real.

    If A and B are sets and f is a function from A to B, then f is a functional relation.
  - Cluster rules:

    If x is an ordinal, then ordsucc x is non empty.
  - Redefinitions:

Mizar Typing System

- Mizar uses some predicates as types.
- "natural" or "non empty" or "Element of X"
- Saying t satisfies a type means a predicate like $P(t, s_1, \ldots, s_m)$ or $\neg P(t, s_1, \ldots, s_m)$ holds.
- The typing rules can essentially be classified as follows:
  - Types of functions f:

    If x and y are reals, then $x + y$ is a real.
  - Hierarchy rules:

    If x is a natural, then x is a real.

    If A and B are sets and f is a function from A to B, then f is a functional relation.
  - Cluster rules:

    If x is an ordinal, then ordsucc x is non empty.
  - Redefinitions:

    If A and B are sets, f is a function from A to B and x is an element of A, then $\mathrm{ap}'\, A\, B\, f\, x = \mathrm{ap}\, f\, x$.

Outline

Example 1

```
Lm1: +infty + +infty = +infty
```

Example 1

```
Lm1: +infty + +infty = +infty
```

- What do $+\infty$, $+$ and $=$ refer to?

Example 1

```
Lm1: +infty + +infty = +infty
```

- What do $+\infty$, $+$ and $=$ refer to?
- In Mizar this would be determined by the environment given in the article's header.

Example 1

```
Lm1: +infty + +infty = +infty
```

- What do $+\infty$, $+$ and $=$ refer to?
- In Mizar this would be determined by the environment given in the article's header.
- We instead have all possibilities from the MML and need to disambiguate.

Example 1

```
Lm1: +infty + +infty = +infty
```

- What do +infty, + and = refer to?
- In Mizar this would be determined by the environment given in the article's header.
- We instead have all possibilities from the MML and need to disambiguate.
- For +infty one choice: `nk1_xxreal_0`.

Example 1

```
Lm1: +infty + +infty = +infty
```

- What do +infty, + and = refer to?
- In Mizar this would be determined by the environment given in the article's header.
- We instead have all possibilities from the MML and need to disambiguate.
- For +infty one choice: nk1_xxreal_0.
- For + two choices are nk1_xxreal_3 and nk10_arytm_3.

Example 1

```
Lm1: +infty + +infty = +infty
```

- What do +infty, + and = refer to?
- In Mizar this would be determined by the environment given in the article's header.
- We instead have all possibilities from the MML and need to disambiguate.
- For +infty one choice: nk1_xxreal_0.
- For + two choices are nk1_xxreal_3 and nk10_arytm_3.
- For = four choices are nr1_hidden, nr4_xboole_0, nr1_relat_1 and nr1_complex1.

Example 1

```
Lm1: +infty + +infty = +infty
```

- What do +infty, + and = refer to?
- In Mizar this would be determined by the environment given in the article's header.
- We instead have all possibilities from the MML and need to disambiguate.
- For +infty one choice: `nk1_xxreal_0`.
- For + two choices are `nk1_xxreal_3` and `nk10_arytm_3`.
- For = four choices are `nr1_hidden`, `nr4_xboole_0`, `nr1_relat_1` and `nr1_complex1`.
- 8 possible readings given by the parser:
- $R(f(i,i),i)$ for one i, two f's and four R's.

Pattern Names vs. Constructors

- Names like `nk1_xxreal_0`, `nk1_xxreal_3` and `nr1_hidden` are "pattern names," not the real symbols which should occur in the real first-order terms and propositions.
- The corresponding constructors are k1_xxreal_0, k1_xxreal_0 and r1_hidden.

Pattern Names vs. Constructors

- Names like `nk1_xxreal_0`, `nk1_xxreal_3` and `nr1_hidden` are "pattern names," not the real symbols which should occur in the real first-order terms and propositions.
- The corresponding constructors are k1_xxreal_0, k1_xxreal_0 and r1_hidden.
- Just drop the "n"? Not quite.

Pattern Names vs. Constructors

- Names like `nk1_xxreal_0`, `nk1_xxreal_3` and `nr1_hidden` are "pattern names," not the real symbols which should occur in the real first-order terms and propositions.
- The corresponding constructors are k1_xxreal_0, k1_xxreal_0 and r1_hidden.
- Just drop the "n"? Not quite.
- `nk10_arytm_3` corresponds to k9_arytm_3.

Pattern Names vs. Constructors

- Names like `nk1_xxreal_0`, `nk1_xxreal_3` and `nr1_hidden` are "pattern names," not the real symbols which should occur in the real first-order terms and propositions.
- The corresponding constructors are k1_xxreal_0, k1_xxreal_0 and r1_hidden.
- Just drop the "n"? Not quite.
- `nk10_arytm_3` corresponds to k9_arytm_3.
- More importantly, the correspondence only holds when certain "pattern guards" hold.

Patterns

- Each pattern name n has a unique corresponding "pattern" of the form

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

or

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) \equiv R(t_1, \ldots, t_l)$$

where the Φ is a set of typing conditions on the variables x_i.

- Patterns are not legitimate formulas because of the name n.

Patterns

- Each pattern name n has a unique corresponding "pattern" of the form

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

or

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) \equiv R(t_1, \ldots, t_l)$$

where the Φ is a set of typing conditions on the variables x_i.

- Patterns are not legitimate formulas because of the name n.
- On the other hand, Φ and the RHS of the conclusion are first-order formulas (or term, for $f(t_1, \ldots, t_l)$).

Patterns

- Each pattern name n has a unique corresponding "pattern" of the form

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

or

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) \equiv R(t_1, \ldots, t_l)$$

where the Φ is a set of typing conditions on the variables x_i.

- Patterns are not legitimate formulas because of the name n.
- On the other hand, Φ and the RHS of the conclusion are first-order formulas (or term, for $f(t_1, \ldots, t_l)$).
- The pattern for n gives necessary "pattern guards" (Φ) and indicates how to "elaborate" the use of a notation name n into a legitimate first-order term or formula.

Patterns

- Each pattern name n has a unique corresponding "pattern" of the form

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

or

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) \equiv R(t_1, \ldots, t_l)$$

where the Φ is a set of typing conditions on the variables x_i.

- Note: If $j > 1$, then $x_1, \ldots, x_{j-1}$ are "implicit arguments."

Elaborating Example 1

```
Lm1: +infty + +infty = +infty
```

- $R(f(i,i),i)$ for various pattern names i, f, R.
- For each choice, we can try to elaborate into a real first-order formula $R'(f'(i',i'),i')$.

Elaborating Example 1

```
Lm1: +infty + +infty = +infty
```

- $R(f(i,i),i)$ for various pattern names i, f, R.
- For each choice, we can try to elaborate into a real first-order formula $R'(f'(i',i'),i')$.
- Each case requires checking pattern guards; sometimes succeed, sometimes fail.

Elaborating Example 1

```
Lm1: +infty + +infty = +infty
```

- $R(f(i,i),i)$ for various pattern names i, f, R.
- For each choice, we can try to elaborate into a real first-order formula $R'(f'(i',i'),i')$.
- Each case requires checking pattern guards; sometimes succeed, sometimes fail.
- IE, not all choices are well-typed (well-typed in the Mizar typing system).

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."

Elaborating Example 1

- i is nk1_xxreal_0 which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be nk1_xxreal_3. The pattern rule for f:

If x and y are extended reals, then $f(x, y)$ is k1_xxreal_3(x, y).

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be `nk1_xxreal_3`. The pattern rule for f:

  If x and y are extended reals, then $f(x, y)$ is k1_xxreal_3(x, y).

- So $f(i, i)$ elaborates to k1_xxreal_3(i', i').

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be `nk1_xxreal_3`. The pattern rule for f:

  If x and y are extended reals, then $f(x, y)$ is k1_xxreal_3(x, y).

- So $f(i, i)$ elaborates to k1_xxreal_3(i', i').
- Take R to be `nr1_hidden`. The pattern rule for R says:

  If x and y are sets, then $R(x, y)$ is r1_hidden(x, y).

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be `nk1_xxreal_3`. The pattern rule for f:

  If x and y are extended reals, then $f(x, y)$ is k1_xxreal_3(x, y).

- So $f(i, i)$ elaborates to k1_xxreal_3(i', i').
- Take R to be `nr1_hidden`. The pattern rule for R says:

  If x and y are sets, then $R(x, y)$ is r1_hidden(x, y).

- The typing system knows every extended real is a set.

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be `nk1_xxreal_3`. The pattern rule for f:

  If x and y are extended reals, then $f(x, y)$ is k1_xxreal_3(x, y).

- So $f(i, i)$ elaborates to k1_xxreal_3(i', i').
- Take R to be `nr1_hidden`. The pattern rule for R says:

  If x and y are sets, then $R(x, y)$ is r1_hidden(x, y).

- The typing system knows every extended real is a set.
- Hence $R(f(i, i), i)$ elaborates to r1_hidden(k1_xxreal_3(i', i'), i').

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be `nk10_arytm_3`. The pattern rule for f:

> *If x and y are natural numbers, then $f(x, y)$ is* k9_arytm_3(x, y).

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be `nk10_arytm_3`. The pattern rule for f:

  If x and y are natural numbers, then $f(x, y)$ is k9_arytm_3(x, y).

- extended real $\not\Rightarrow$ natural number.

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be `nk10_arytm_3`. The pattern rule for f:

  If x and y are natural numbers, then $f(x, y)$ is k9_arytm_3(x, y).

- extended real $\not\Rightarrow$ natural number.
- Stop the elaboration in this case and rule out this case.

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be `nk10_arytm_3`. The pattern rule for f:

  If x and y are natural numbers, then $f(x, y)$ is k9_arytm_3(x, y).

- extended real $\not\Rightarrow$ natural number.
- Stop the elaboration in this case and rule out this case.
- Slight problem: sometimes the checker is too weak, so we may assume some pattern guards and continue.

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be `nk10_arytm_3`. The pattern rule for f:

  If x and y are natural numbers, then $f(x, y)$ is k9_arytm_3(x, y).

- extended real $\not\Rightarrow$ natural number.
- Stop the elaboration in this case and rule out this case.
- Slight problem: sometimes the checker is too weak, so we may assume some pattern guards and continue.
- Rule out an elaboration if too many pattern guards need to be assumed.

Elaborating Example 1

- i is `nk1_xxreal_0` which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be `nk10_arytm_3`. The pattern rule for f:

  If x and y are natural numbers, then $f(x, y)$ is k9_arytm_3(x, y).

- extended real $\not\Rightarrow$ natural number.
- Stop the elaboration in this case and rule out this case.
- Slight problem: sometimes the checker is too weak, so we may assume some pattern guards and continue.
- Rule out an elaboration if too many pattern guards need to be assumed.
- In Example 1, only one parse is well-typed with no assumed guards.

Elaborating Example 1

- i is nk1_xxreal_0 which elaborates to k1_xxreal_0 (i').
- Type of i' is "extended real."
- Take f to be nk10_arytm_3. The pattern rule for f:

  If x and y are natural numbers, then $f(x, y)$ is k9_arytm_3(x, y).

- extended real $\not\Rightarrow$ natural number.
- Stop the elaboration in this case and rule out this case.
- Slight problem: sometimes the checker is too weak, so we may assume some pattern guards and continue.
- Rule out an elaboration if too many pattern guards need to be assumed.
- In Example 1, only one parse is well-typed with no assumed guards.
- In the 7 other parses, at least 40% of checked pattern guards must be assumed.

Example 2 (Implicit Arguments)

- In the MML, function application is defined twice, using an infix dot.

Example 2 (Implicit Arguments)

- In the MML, function application is defined twice, using an infix dot.
- The first dot depends on two arguments, the function and its argument:

```
definition
  let f be Function;
  let x be object ;
  func f . x -> set means :Def2: :: FUNCT_1:def 2
  [x,it] in f if x in dom f
  otherwise it = {} ;
  existence
  proof end;
  uniqueness by Def1;
  consistency ;
end;
```

Example 2 (Implicit Arguments)

- In the MML, function application is defined twice, using an infix dot.
- The second dot is a redefinition depending on four arguments (the domain, codomain, function and the argument):

```
definition
  let C be non empty set ;
  let D be set ;
  let f be Function of C,D;
  let c be Element of C;
  :: original: .
  redefine func f . c -> Element of D;
  coherence
  proof end;
end;
```

Example 2 (Implicit Arguments)

- In the MML, function application is defined twice, using an infix dot.
- The second dot is a redefinition depending on four arguments (the domain, codomain, function and the argument):

```
definition
  let C be non empty set ;
  let D be set ;
  let f be Function of C,D;
  let c be Element of C;
  :: original: .
  redefine func f . c -> Element of D;
  coherence
  proof end;
end;
```

- C, D are implicit in the pattern for the second:

If C is a nonempty set, D is a set, f is a function from C to D and c is in C, then nk3_funct_2(f, c) is k3_funct_2(C, D, f, c).

Example 2 (Implicit Arguments)

- Consider $\mathit{sin}.\mathit{sin}.0 = 0$
- One possible parse takes *sin* to be `nk17_sin_cos`, dot to be the second function application and 0 to be 0.

Example 2 (Implicit Arguments)

- Consider $sin.sin.0 = 0$
- One possible parse takes *sin* to be `nk17_sin_cos`, dot to be the second function application and 0 to be 0.
- The pattern for `nk17_sin_cos` means it elaborates to k16_sin_cos which has as type "function from reals to reals."

Example 2 (Implicit Arguments)

- Consider $sin.sin.0 = 0$
- One possible parse takes *sin* to be `nk17_sin_cos`, dot to be the second function application and 0 to be 0.
- The pattern for `nk17_sin_cos` means it elaborates to k16_sin_cos which has as type "function from reals to reals."
- Using this type and the pattern for application we can recover the domain and codomain (the set of reals R) when elaborating both occurrences of application.

Example 2 (Implicit Arguments)

- Consider $sin.sin.0 = 0$
- One possible parse takes *sin* to be `nk17_sin_cos`, dot to be the second function application and 0 to be 0.
- The pattern for `nk17_sin_cos` means it elaborates to k16_sin_cos which has as type "function from reals to reals."
- Using this type and the pattern for application we can recover the domain and codomain (the set of reals R) when elaborating both occurrences of application.
- Since 0 and k3_funct_2$(R, R,$ k16_sin_cos$, 0)$ can be checked to be elements of R, the parse can be fully elaborated as

$$\text{r1\_hidden}(\text{k3\_funct\_2}(R, R, \text{k16\_sin\_cos}, \text{k3\_funct\_2}(R, R, \text{k16\_sin\_cos}, 0)), 0)$$

Elaboration and Checking

- Two languages L_1 and L_2.
- Only L_1 has a sensible semantics. Each name in L_2 can be "elaborated" into an L_1 name.
- Elaborate proposition/term in language L_2 to one in language L_1.

$$s_2 \blacktriangleright s_1$$

- Doing this may require filling in implicit arguments.
- Result should be "well-typed" in the Mizar sense (type checking).
- The two processes interact since the way to obtain implicit arguments is by looking at types computed so far.

Outline

Checking Algorithm

- Goal: Compute all types of $f(t_1, \ldots, t_l)$.

Checking Algorithm

- Goal: Compute all types of $f(t_1, \ldots, t_l)$.
- Compute all types of each argument t_i.

Checking Algorithm

- Goal: Compute all types of $f(t_1, \ldots, t_l)$.
- Compute all types of each argument t_i.
- Use the function typing rules for f to compute initial types for $f(t_1, \ldots, t_l)$.

Checking Algorithm

- Goal: Compute all types of $f(t_1, \ldots, t_l)$.
- Compute all types of each argument t_i.
- Use the function typing rules for f to compute initial types for $f(t_1, \ldots, t_l)$.
- Each time a type is computed,
  - "widen" the type applying hierarchy rules and
  - "round up" cluster rules,
  - working modulo redefinition rules.

Elaboration Algorithm

- Given $n(s_j, \ldots, s_k)$ for a pattern function n.

Elaboration Algorithm

- Given $n(s_j, \ldots, s_k)$ for a pattern function n.
- Elaborate $s_j, \ldots, s_k$ computing s'_i where $s_i \blacktriangleright s'_i$ and compute types for each s'_i.

Elaboration Algorithm

- Given $n(s_j, \ldots, s_k)$ for a pattern function n.
- Elaborate $s_j, \ldots, s_k$ computing s'_i where $s_i \blacktriangleright s'_i$ and compute types for each s'_i.
- Look up pattern rule for n:

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

Elaboration Algorithm

- Given $n(s_j, \ldots, s_k)$ for a pattern function n.
- Elaborate $s_j, \ldots, s_k$ computing s'_i where $s_i \blacktriangleright s'_i$ and compute types for each s'_i.
- Look up pattern rule for n:

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

- Let σ be the substitution $\sigma(x_i) = s'_i$ for $j \leq i \leq k$.

Elaboration Algorithm

- Given $n(s_j, \ldots, s_k)$ for a pattern function n.
- Elaborate $s_j, \ldots, s_k$ computing s'_i where $s_i \blacktriangleright s'_i$ and compute types for each s'_i.
- Look up pattern rule for n:

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

- Let σ be the substitution $\sigma(x_i) = s'_i$ for $j \leq i \leq k$.
- For each pattern guard in Φ of the form $P(s'_i, \ldots)$ match against known types of s'_i. This may extend σ (computing implicit arguments).

Elaboration Algorithm

- Given $n(s_j, \ldots, s_k)$ for a pattern function n.
- Elaborate $s_j, \ldots, s_k$ computing s'_i where $s_i \blacktriangleright s'_i$ and compute types for each s'_i.
- Look up pattern rule for n:

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

- Let σ be the substitution $\sigma(x_i) = s'_i$ for $j \leq i \leq k$.
- For each pattern guard in Φ of the form $P(s'_i, \ldots)$ match against known types of s'_i. This may extend σ (computing implicit arguments).
- Each time a new $\sigma(x_i)$ is determined, compute its types.

Elaboration Algorithm

- Given $n(s_j, \ldots, s_k)$ for a pattern function n.
- Elaborate $s_j, \ldots, s_k$ computing s'_i where $s_i \blacktriangleright s'_i$ and compute types for each s'_i.
- Look up pattern rule for n:

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

- Let σ be the substitution $\sigma(x_i) = s'_i$ for $j \leq i \leq k$.
- For each pattern guard in Φ of the form $P(s'_i, \ldots)$ match against known types of s'_i. This may extend σ (computing implicit arguments).
- Each time a new $\sigma(x_i)$ is determined, compute its types.
- In the end, if $\sigma(x_i)$ is not defined for some $1 \leq i < j$, then fail.

Elaboration Algorithm

- Given $n(s_j, \ldots, s_k)$ for a pattern function n.
- Elaborate $s_j, \ldots, s_k$ computing s'_i where $s_i \blacktriangleright s'_i$ and compute types for each s'_i.
- Look up pattern rule for n:

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

- Let σ be the substitution $\sigma(x_i) = s'_i$ for $j \leq i \leq k$.
- For each pattern guard in Φ of the form $P(s'_i, \ldots)$ match against known types of s'_i. This may extend σ (computing implicit arguments).
- Each time a new $\sigma(x_i)$ is determined, compute its types.
- In the end, if $\sigma(x_i)$ is not defined for some $1 \leq i < j$, then fail.
- Otherwise check which pattern guards are known. Assume those that are unknown.

Elaboration Algorithm

- Given $n(s_j, \ldots, s_k)$ for a pattern function n.
- Elaborate $s_j, \ldots, s_k$ computing s'_i where $s_i \blacktriangleright s'_i$ and compute types for each s'_i.
- Look up pattern rule for n:

$$\forall x_1, \cdots, x_k.\Phi \rightarrow n(x_j, \ldots, x_k) = f(t_1, \ldots, t_l)$$

- Let σ be the substitution $\sigma(x_i) = s'_i$ for $j \leq i \leq k$.
- For each pattern guard in Φ of the form $P(s'_i, \ldots)$ match against known types of s'_i. This may extend σ (computing implicit arguments).
- Each time a new $\sigma(x_i)$ is determined, compute its types.
- In the end, if $\sigma(x_i)$ is not defined for some $1 \leq i < j$, then fail.
- Otherwise check which pattern guards are known. Assume those that are unknown.
- Return $n(s_j, \ldots, s_k) \blacktriangleright \sigma(f(t_1, \ldots, t_l))$ and compute all types of $\sigma(f(t_1, \ldots, t_l))$.

Outline

Elaborating Pattern Based Mizar Theorems

- Input: About 60K Mizar Theorems in Pattern Representation
- Output: Constructor Based Version or Failure or Timeout (20s)

Elaborating Pattern Based Mizar Theorems

- Input: About 60K Mizar Theorems in Pattern Representation
- Output: Constructor Based Version or Failure or Timeout (20s)
- About 95% elaborated with no assumed pattern guards

Elaborating Pattern Based Mizar Theorems

- Input: About 60K Mizar Theorems in Pattern Representation
- Output: Constructor Based Version or Failure or Timeout (20s)
- About 95% elaborated with no assumed pattern guards
- Another 2% elaborated with some assumed pattern guards

Elaborating Pattern Based Mizar Theorems

- Input: About 60K Mizar Theorems in Pattern Representation
- Output: Constructor Based Version or Failure or Timeout (20s)
- About 95% elaborated with no assumed pattern guards
- Another 2% elaborated with some assumed pattern guards
- Roughly 2% fail to fill in some implicit arguments

Elaborating Pattern Based Mizar Theorems

- Input: About 60K Mizar Theorems in Pattern Representation
- Output: Constructor Based Version or Failure or Timeout (20s)
- About 95% elaborated with no assumed pattern guards
- Another 2% elaborated with some assumed pattern guards
- Roughly 2% fail to fill in some implicit arguments
- Roughly 1% time out

Correctness of Elaboration

- In about 71% of elaborated cases, a fof constructor version can be produced.
- Is this fof constructor version equivalent to the constructor version of the corresponding theorem in the MPTP?

Correctness of Elaboration

- In about 71% of elaborated cases, a fof constructor version can be produced.
- Is this fof constructor version equivalent to the constructor version of the corresponding theorem in the MPTP?
- Yes, and E can prove this equivalence in roughly 98% of cases.

Testing with the Parser

- The training data for the parser contains the pattern based versions as a parse tree (including presentation information such as literals like `for` and `implies`).
- We can directly send this parse tree to the elaborator, have the elaborator extract a pattern representaion as a formula and attempt to elaborate it to a constructor based formula.

Testing with the Parser

- The training data for the parser contains the pattern based versions as a parse tree (including presentation information such as literals like `for` and `implies`).
- We can directly send this parse tree to the elaborator, have the elaborator extract a pattern representaion as a formula and attempt to elaborate it to a constructor based formula.
- Elaborator mysteriously only called for 36K (about 59%) examples.

Testing with the Parser

- The training data for the parser contains the pattern based versions as a parse tree (including presentation information such as literals like `for` and `implies`).
- We can directly send this parse tree to the elaborator, have the elaborator extract a pattern representaion as a formula and attempt to elaborate it to a constructor based formula.
- Elaborator mysteriously only called for 36K (about 59%) examples.
- There are constructions in the training set parse trees the code does not recognize (like "the carrier of X")

Testing with the Parser

- The training data for the parser contains the pattern based versions as a parse tree (including presentation information such as literals like `for` and `implies`).
- We can directly send this parse tree to the elaborator, have the elaborator extract a pattern representaion as a formula and attempt to elaborate it to a constructor based formula.
- Elaborator mysteriously only called for 36K (about 59%) examples.
- There are constructions in the training set parse trees the code does not recognize (like "the carrier of X")
- 55% of these 36K elaborated with no assumed pattern guards

Testing with the Parser

- The training data for the parser contains the pattern based versions as a parse tree (including presentation information such as literals like `for` and `implies`).
- We can directly send this parse tree to the elaborator, have the elaborator extract a pattern representaion as a formula and attempt to elaborate it to a constructor based formula.
- Elaborator mysteriously only called for 36K (about 59%) examples.
- There are constructions in the training set parse trees the code does not recognize (like "the carrier of X")
- 55% of these 36K elaborated with no assumed pattern guards
- 75% of these 36K elaborated with assumed pattern guards

Testing with the Parser

- The training data for the parser contains the pattern based versions as a parse tree (including presentation information such as literals like `for` and `implies`).
- We can directly send this parse tree to the elaborator, have the elaborator extract a pattern representaion as a formula and attempt to elaborate it to a constructor based formula.
- Elaborator mysteriously only called for 36K (about 59%) examples.
- There are constructions in the training set parse trees the code does not recognize (like "the carrier of X")
- 55% of these 36K elaborated with no assumed pattern guards
- 75% of these 36K elaborated with assumed pattern guards
- 25% fail to fill in some implicit arguments

Testing with the Parser

- The training data for the parser contains the pattern based versions as a parse tree (including presentation information such as literals like `for` and `implies`).
- We can directly send this parse tree to the elaborator, have the elaborator extract a pattern representaion as a formula and attempt to elaborate it to a constructor based formula.
- Elaborator mysteriously only called for 36K (about 59%) examples.
- There are constructions in the training set parse trees the code does not recognize (like "the carrier of X")
- 55% of these 36K elaborated with no assumed pattern guards
- 75% of these 36K elaborated with assumed pattern guards
- 25% fail to fill in some implicit arguments
- When it works though, it gives a meaningful formula that can be sent to a theorem prover.

Planned Parser Tests

- We planned to do 100 fold crossvalidation on the training set.
- Current tests however show that the integration with the elaborator is not yet working well enough for the results to be meaningful.
- Without calling the elaborator, the parser can obtain the correct parse within the top 20 parses roughly 60% of the time.

Planned Parser Tests

- We planned to do 100 fold crossvalidation on the training set.
- Current tests however show that the integration with the elaborator is not yet working well enough for the results to be meaningful.
- Without calling the elaborator, the parser can obtain the correct parse within the top 20 parses roughly 60% of the time.
- With the elaborator, the correct parse is only found about 2% of the time.

Planned Parser Tests

- We planned to do 100 fold crossvalidation on the training set.
- Current tests however show that the integration with the elaborator is not yet working well enough for the results to be meaningful.
- Without calling the elaborator, the parser can obtain the correct parse within the top 20 parses roughly 60% of the time.
- With the elaborator, the correct parse is only found about 2% of the time.
- Here is one example that works with the elaborator but not without:

```
for seq being Real_Sequence holds ( seq is
  non-zero iff for n being Element of NAT
              holds seq .  n <> 0 )
```

Outline

Conclusion

- MML in various representations.
- Probabilistic parser to learn from parse tree representations and then parse underlying text.
- Conversion of parse trees to pattern representations
- Elaborator/type-checker to obtain constructor representations
- Constructor representations are formulas which can meaningfully be used with a theorem prover.

Conclusion

- MML in various representations.
- Probabilistic parser to learn from parse tree representations and then parse underlying text.
- Conversion of parse trees to pattern representations
- Elaborator/type-checker to obtain constructor representations
- Constructor representations are formulas which can meaningfully be used with a theorem prover.
- We have all the pieces to go from informal to formal, at least with Mizar data

Conclusion

- MML in various representations.
- Probabilistic parser to learn from parse tree representations and then parse underlying text.
- Conversion of parse trees to pattern representations
- Elaborator/type-checker to obtain constructor representations
- Constructor representations are formulas which can meaningfully be used with a theorem prover.
- We have all the pieces to go from informal to formal, at least with Mizar data
- ...though some of the pieces are still being put together

Outline

Introduction

Parsing

Mizar Typing System

Examples

Algorithms

Results

Conclusion

References

References

- Urban. MoMM - Fast Interreduction and Retrieval in Large Libraries of Formalized Mathematics. IJAIT 2006
- Nelson, Oppen. Fast Decision Procedures Based on Congruence Closure. JACM 1980
- Xi, Pfenning. Dependent Types in Practical Programming. POPL 1999
- de Moura, Avigad, Kong, Roux. Elaboration in Dependent Type Theory. 2015
- Ferreira, Pientka. Bidirectional Elaboration of Dependently Typed Programs. PPDP 2014