Overcoming Performance Barriers: efficient proof search in logical frameworks

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada
Outline

- Logical frameworks and applications
- Efficient proof search in logical frameworks
  - Optimizing higher-order unification
  - Higher-order term indexing
- Conclusion and future work
Logical frameworks

• Meta-languages for deductive systems
  • High-level specification (e.g. logics, type systems)
  • Direct implementations (e.g. proof search, type checking)
  • Meta-reasoning (e.g. cut elim., type preservation)

• Examples:
  λProlog[Nadathur’99], Twelf[Pf’99], Isabelle[Paulson86]

• Other higher-order systems: Coq, PVS, Nuprl, HOL, ...

Overcoming Performance Barriers: – p.3/38
Higher-order logic programming

- Higher-order data-types: dependently typed $\lambda$-calculus
Higher-order logic programming

- Higher-order data-types: dependently typed $\lambda$-calculus
- Dynamic program clauses: Clauses may contain nested universal quantifiers and implications
Higher-order logic programming

- Higher-order data-types: dependently typed $\lambda$-calculus
- Dynamic program clauses: Clauses may contain nested universal quantifiers and implications
- Result of query execution: Evidence for a proof together with answer substitution
Higher-order logic programming

- Higher-order data-types: dependently typed $\lambda$-calculus
- Dynamic program clauses: Clauses may contain nested universal quantifiers and implications
- Result of query execution: Evidence for a proof together with answer substitution
- Theoretical foundation based on uniform proofs [Miller et. al. 91], [Pf’91]
Higher-order logic programming

- Higher-order data-types: dependently typed $\lambda$-calculus
- Dynamic program clauses: Clauses may contain nested universal quantifiers and implications
- Result of query execution: Evidence for a proof together with answer substitution
- Theoretical foundation based on uniform proofs [Miller et. al. 91], [Pf’91]
- Extensions to tabled higher-order logic programming [Pie’03, Pie’05]
Example

- **Object logic: First-order logic formula**

  \[ A ::= P \mid A \supset A \mid A \lor A \mid \neg A \mid \forall x. A \mid \exists x. A \mid \ldots \]

- **Specifying equivalence preserving transformations**

- **Sample rules:**

  \[ A \supset B \iff \neg A \lor B \]

  \[ \forall x. (A(x) \lor B) \iff (\forall x. A(x)) \lor B \]

  \[ \forall x. (A(x) \supset B) \iff (\exists x. A(x)) \supset B \]

  if \( x \) is not free in \( B \)
Based on higher order abstract syntax:

- $i$ : type.
- $o$ : type
- neg : $o \rightarrow o$
- imp : $o \rightarrow o \rightarrow o$.
- all : $(i \rightarrow o) \rightarrow o$.
- or : $o \rightarrow o \rightarrow o$.
- exists : $(i \rightarrow o) \rightarrow o$.

Transforming propositions:

$$A \supset B \iff \neg A \lor B$$

eq_imp: $\text{eq} (A \text{ imp } B) \iff ((\text{not } A) \text{ or } B)$
Specication in LF

• Based on higher order abstract syntax:

\[
\begin{align*}
i & : \text{type.} & o & : \text{type} \\
neg & : o \rightarrow o \\
\text{imp} & : o \rightarrow o \rightarrow o. & \text{all} & : (i \rightarrow o) \rightarrow o. \\
\text{or} & : o \rightarrow o \rightarrow o. & \text{exists} & : (i \rightarrow o) \rightarrow o.
\end{align*}
\]

• Transforming propositions:

\[
\begin{align*}
\forall x. (A(x) \supset B) & \iff (\exists x. A(x)) \supset B \\
eq\text{all} & : \text{eq} \ (\text{all} \ (\lambda x. (A \ x) \ \text{imp} \ B)) \quad ((\text{exists} \ (\lambda x. A \ x)) \ \text{imp} \ B).
\end{align*}
\]
Specification in LF

- Based on higher order abstract syntax:

  \[
  \begin{align*}
  i & \ : \ type. \quad o & \ : \ type \\
  \text{neg} & \ : \ o \rightarrow o \\
  \text{imp} & \ : \ o \rightarrow o \rightarrow o. \quad \text{all} & \ : \ (i \rightarrow o) \rightarrow o. \\
  \text{or} & \ : \ o \rightarrow o \rightarrow o. \quad \text{exists} & \ : \ (i \rightarrow o) \rightarrow o.
  \end{align*}
  \]

- Transforming propositions:

  \[
  \forall x. (A(x) \supset B) \iff (\exists x. A(x)) \supset B
  \]

  \[
  \text{eq}_{\text{all}}: \text{eq} \quad (\text{all} \ (\lambda x. (A \ x) \ \text{imp} \ B)) \quad (\text{exists} \ (\lambda x. A \ x)) \ \text{imp} \ B).
  \]

- A: i \rightarrow o and B: o are **meta-variables**
  
  also sometimes called **existential variables or logic variables**
Application: certified code

- Foundational proof-carrying code: [Appel, Felty 00]
- Temporal-logic proof carrying code: [Bernard, Lee 02]
- Foundational typed assembly language: [Crary 03]
- Distributed access control: [Bauer, Reiter’05]
Large-scale applications

- Typical code size: 70,000 – 100,000 lines includes data-type definitions and proofs
- Higher-order logic program: 5,000 lines
- Over 600 – 700 clauses
Application: certified code

Special-purpose logical frameworks:

- Efficient representation and validation of proofs [Necula, Lee'98] [Reed'04]
- Proof checking via “higher-order” logic programming [Necula’01], [Wu’03]
Neglected aspect: language we write programs in

We need tools to

- Model and specify programming languages
- Experiment easily with language extensions
- Mechanically check their meta-theoretic properties

POPLmark Challenge [Pierce et al 05]
“Mechanically check every POPL paper by 2010!”

Logical framework allows us to represent, execute, and reason about formal systems.
State of the art

• Logical frameworks are widely used.
• Many challenges remain:
  • Higher-order systems are not efficient enough in practice.
  • Complexity of higher-order issues poorly understood.
  • Higher-order systems lack automatic support.
  • ...
State of the art

• Logical frameworks are widely used.
• Many challenges remain:
  • Higher-order systems are not efficient enough in practice.
  • Complexity of higher-order issues poorly understood.
  • Higher-order systems lack automatic support.
  • ...
This talk

Eliminating some performance problems

- Optimizing higher-order unification
- Higher-order term indexing

This is a significant step towards efficient proof search in logical frameworks
Outline

- Logical frameworks and applications
- Efficient proof search in logical frameworks
  - Optimizing higher-order unification
  - Higher-order term indexing
- Conclusion and future work
Outline

- Logical frameworks and applications
- Efficient proof search in logical frameworks
  - Optimizing higher-order unification
  - Higher-order term indexing
- Conclusion and future work
“For any programming language to be practical, basic operations should take constant time. Unification ... may be thought of as the basic operation...” [Sicstus Prolog Manual]
“For any programming language to be practical, basic operations should take constant time. Unification ... may be thought of as the basic operation...” [Sicstus Prolog Manual]

Higher-order unification is undecidable!
“For any programming language to be practical, basic operations should take constant time. Unification ... may be thought of as the basic operation...” [Sicstus Prolog Manual]

Higher-order unification is undecidable!

For decidable fragment [Miller91, Pfenning91]: at best linear [Qian93]!
Basic operation: unification

- **Example 1:**
  
  \[
  \begin{align*}
  &\text{eq } (A \text{ imp } B) ((\text{not } A) \text{ or } B) \quad \text{Success} \\
  &\text{eq } (p \text{ imp } q) ((\text{not } C) \text{ or } q) \quad A = p, \quad B = q, \quad C = A
  \end{align*}
  \]
Basic operation: unification

- Example 1:
  \[
  \begin{align*}
  &eq \ (A \ imp \ B) \ ((not \ A) \ or \ B) \quad \text{Success} \\
  &eq \ (p \ imp \ q) \ ((not \ C) \ or \ q) \quad A = p, \quad B = q, \quad C = A
  \end{align*}
  \]

- Example 2:
  \[
  \begin{align*}
  &eq \ (A \ imp \ B) \ ((not \ A) \ or \ B) \quad \text{Failure(occurs-check!)} \\
  &eq \ C \ ((not \ C) \ or \ q) \quad C = (A \ imp \ B), \quad A = C, \quad B = q
  \end{align*}
  \]
Basic operation: unification

• Example 1:
  \[
  \text{eq } (A \text{ imp } B) \ (\text{not } A) \text{ or } B
  \]
  \[
  \text{eq } (p \text{ imp } q) \ (\text{not } C) \text{ or } q
  \]
  Success
  A = p, \ B = q, \ C = A

• Example 2:
  \[
  \text{eq } (A \text{ imp } B) \ (\text{not } A) \text{ or } B
  \]
  \[
  \text{eq } C \ (\text{not } C) \text{ or } q
  \]
  Failure(occurs-check!)
  C = (A \text{ imp } B),
  A = C, \ B = q

• Occurs check is expensive!
Basic operation: unification

• Example 1:
  \[
  \begin{align*}
  &\text{eq } (A \imp B) \text{ } ((\neg A) \text{ or } B) \quad \text{Success} \\
  &\text{eq } (p \imp q) \text{ } ((\neg C) \text{ or } q) \quad A = p, \quad B = q, \quad C = A
  \end{align*}
  \]

• Example 2:
  \[
  \begin{align*}
  &\text{eq } (A \imp B) \text{ } ((\neg A) \text{ or } B) \quad \text{Failure(occurs-check!)} \\
  &\text{eq } C \text{ } ((\neg C) \text{ or } q) \quad C = (A \imp B), \quad A = C, \quad B = q
  \end{align*}
  \]

• Occurs check is expensive!

• No occurs check is necessary if every meta-variable occurs only once!
Higher-order pattern unification

- Meta-variables must be applied to some distinct bound variables

\[(\forall x. ((A \ x) \imp B)) \rightarrow \text{ok} \quad (\exists x. A \ x) \imp B \rightarrow \text{not ok!}\]
Higher-order pattern unification

- Meta-variables must be applied to some distinct bound variables
  
  \[(\forall \lambda x. ((A x) \implies B)) \quad \text{– ok} \quad \quad ((C T) \implies B) \quad \text{– not ok!}\]

- Closed instantiation for meta-variables!

\[
\begin{align*}
\text{eq} \quad (\forall \lambda y. ((p y) \implies (p y)) \implies q) \quad C \\
\quad \quad \equiv \\
\text{eq} \quad (\forall \lambda x. (A x) \implies B) \quad ((\exists \lambda x. A x) \implies B)
\end{align*}
\]
Higher-order pattern unification

• Meta-variables must be applied to some distinct bound variables
  
  \[(\forall \lambda x. ((A x) \implies B)) \, \text{– ok} \quad \quad ((C \, T) \implies B) \, \text{– not ok!}\]

• Closed instantiation for meta-variables!

  \[
  \begin{align*}
  &\text{eq } (\forall \lambda y. \, ((p \, y) \implies (p \, y)) \implies q) \quad C \\
  \end{align*}
  \]

  \[
  \begin{align*}
  \Rightarrow \\
  &\text{eq } (\forall \lambda x. \, (A \, x) \implies B) \quad ((\exists \lambda x. \, A \, x) \implies B) \\
  \end{align*}
  \]

• Solution:

  \[
  \begin{align*}
  A &= \lambda z. \, (p \, z) \implies (p \, z) \\
  B &= q \\
  C &= ((\exists \lambda x. \, A \, x) \implies B) \\
  &= (\implies (\exists \lambda x. \, \implies (p \, x) \, (p \, x))) \, q
  \end{align*}
  \]
Higher-order pattern unification

- Meta-variables must be applied to some distinct bound variables
  \[(\text{all } \lambda x. ((A \ x) \text{ imp } B)) \quad \text{– ok} \quad ((C \ T) \text{ imp } B) \quad \text{– not ok!}\]

- Closed instantiation for meta-variables?
  \[\text{eq } (\text{all } \lambda y. \ ((p \ y) \text{ imp } (p \ y)) \quad \text{imp } (p \ y)) \quad C \]
  \[\overset{=} \quad \text{eq } (\text{all } \lambda x. \ (A \ x) \quad \text{imp } B) \quad ((\text{exists } \lambda x. \ A \ x) \text{ imp } B)\]
**Higher-order pattern unification**

- Meta-variables must be applied to some distinct bound variables
  
  \[(\forall \lambda x. (((A x) \text{ imp } B)) \rightarrow \text{ ok}) \quad ((C T) \text{ imp } B) \rightarrow \text{ not ok!}\]

- Closed instantiation for meta-variables?
  
  \[\text{eq } (\forall \lambda y. (((p y) \text{ imp } (p y)) \text{ imp } (p y)) \quad C \quad \Rightarrow \quad \]

  \[\text{eq } (\forall \lambda x. (A x) \text{ imp } B) \quad ((\exists \lambda x. A x) \text{ imp } B)\]

- Failure
  
  \[A = \lambda z. (p z) \text{ imp } (p z)\]
  
  \[B = ? \quad \text{There is no closed instantiation for } B!\]
  
  \[C = \ldots\]
Subtle issues due to bound variables

• Which bound variables are allowed to occur in a term that instantiates a meta-variable?
  – A depends on bound variable x
  – B does not depend on bound variable x
  – Computing dependencies may be expensive!

No check is necessary, if meta-variable depends on all distinct bound variables.
Subtle issues due to bound variables

• Which bound variables are allowed to occur in a term that instantiates a meta-variable?
  – A depends on bound variable x
  – B does not depend on bound variable x
  – Computing dependencies may be expensive!

• No check is necessary, if meta-variable depends on all distinct bound variables.
Linearization

• Linear terms:
  - every meta-variable occurs only once
  - every meta-variable depends on all distinct bound variables
Linearization

- Linear terms:
  - every meta-variable occurs only once
  - every meta-variable depends on all distinct bound variables

- Every clause head is transformed into a linear term and variable definitions
Linearization

- Linear terms:
  - every meta-variable occurs only once
  - every meta-variable depends on all distinct bound variables

- Every clause head is transformed into a linear term and variable definitions

Example:

\[
\begin{align*}
\text{eq } (A \text{ imp } B) & \quad ((\text{not } A) \text{ or } B) \\
\iff \\
\text{eq } (A \text{ imp } B) & \quad ((\text{not } A') \text{ or } B') \quad \text{and } \quad A' = A \quad \text{and } \quad B' = B
\end{align*}
\]
Linearization

- Linear terms:
  - every meta-variable occurs only once
  - every meta-variable depends on all distinct bound variables

- Every clause head is transformed into a linear term and variable definitions

- Example:

  \[ \text{eq} \ (\forall x. \ (A x) \imp B) \ (\forall x. \ (A x) \imp B) \]

  \[ \iff \]

  \[ \text{eq} \ (\forall x. \ (A x) \imp (B' x)) \ (\forall x. \ (A' x) \imp B) \]

  \[ A' = A \quad \text{and} \quad \forall x. \ (B' x) \equiv B \]
Why does linearization work?

- Linearization is performed statically.
Why does linearization work?

- Linearization is performed statically.
- Many problems are already linear.
  constant time assignment algorithm
Why does linearization work?

- Linearization is performed statically.
- Many problems are already linear.
  constant time assignment algorithm
- Unification often fails.
  Failure can be very expensive in higher-order unification,
  even in the decidable fragment.
## Foundational PCC

<table>
<thead>
<tr>
<th>example</th>
<th>standard</th>
<th>opt</th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>mul2</td>
<td>9.52 sec</td>
<td>5.51 sec</td>
<td>42.86%</td>
</tr>
<tr>
<td>div2</td>
<td>153.61 sec</td>
<td>121.96 sec</td>
<td>20.63%</td>
</tr>
<tr>
<td>pack</td>
<td>1075.61 sec</td>
<td>197.07 sec</td>
<td>81.65%</td>
</tr>
<tr>
<td>divx</td>
<td>1133.15 sec</td>
<td>333.69 sec</td>
<td>70.50%</td>
</tr>
<tr>
<td>listsum</td>
<td>∞</td>
<td>1073.33 sec</td>
<td>100%</td>
</tr>
</tbody>
</table>

∞ = process does not terminate in 6h

Intel Pentium 1.6GHz, RAM 256MB,
SML New Jersey 110, Twelf 1.4.
Evaluation

- Performance improvement is substantial
  20% – 82% runtime improvement; in some case 100%!
  - 63% of the time there are no variable defs.
  - 80% of the calls to unification failed

- Benchmarks (simply typed):
  - Meta-interpreter for linear ordered logic: 60%
  - Classical natural deduction (NK): 42%

- Benchmarks (dependently typed):
  - Compiler translations: 99.95%, in some cases 100%
  - Translating proofs into cut-free proofs: 43% - 52%
Contribution and related work

- Foundation for meta-variables based on modal logic (joint work with F. Pfenning) (CADE’03)
  - Extends earlier work by [Dowek et al. 95]
  - Contextual modal type theory and applications
    (joint work with A. Nanevski, F. Pfenning, 2005)

- Related work: λProlog (Teyjus-compiler) [Nadathur, Mitchell 99]
  - General higher-order unification
    (highly non-deterministic)
  - WAM with special higher-order support
Optimizing unification further

- Eliminating redundant type arguments [IJCAR’06]
  - Dependently typed terms have implicit type arguments
  - Some implicit type arguments in a term $M$ are uniquely determined by the overall type of $M$.
  - These implicit arguments can be skipped during unification!

- Early empirical study [Michaylov,Pfenning’92]
Experiments and evaluation

• Compiler translation:
  • Substantial number of redundant type arguments (up to 1496)
  • Substantial size of skipped arguments (av 30, max 185)
  • Run-time improvement: 11.19% - 21.87%

• Proof translations:
  • Substantial number of redundant type arguments (up to 264387)
  • Size of skipped arguments (av 7)
  • Run-time improvement: 3% - 10%
Contribution and related work

- Performance improvement up to 20%
- Numerous redundant type arguments
- Theoretical justification [IJCAR06]
- Related Work: \( \lambda \)-Prolog: redundant type arguments due to polymorphism [Nadathur, Qi’05]
  - incorporated into the WAM
  - no experimental comparison
Outline

• Logical frameworks and applications
• Efficient proof search in logical frameworks
  - Optimizing higher-order unification
  - Higher-order term indexing
• Conclusion and future work
• Logical frameworks and applications
• Efficient proof search in logical frameworks
  - Optimizing higher-order unification
  - Higher-order term indexing
• Conclusion and future work
“...an automated reasoning program’s rate of drawing conclusions falls off sharply both with time and with an increase in the size of the database of retained information.” [Wos92]
“...an automated reasoning program’s rate of drawing conclusions falls off sharply both with time and with an increase in the size of the database of retained information.” [Wos92]
“...an automated reasoning program’s rate of drawing conclusions falls off sharply both with time and with an increase in the size of the database of retained information.” [Wos92]
Indexing

Set of terms

\[ \text{eq} \quad (\forall x. ((A \cdot x) \lor B)) \quad ((\forall x. A \cdot x) \lor B) \]
\[ \text{eq} \quad (A \rightarrow B) \quad ((\neg A) \lor B) \]
\[ \text{eq} \quad (\neg (A \land B)) \quad ((\neg A) \lor (\neg B)) \]

How can we efficiently store and retrieve data?
Indexing

Set of terms

\[
\begin{align*}
\text{eq} & \quad (\forall x. ((A \ x) \lor B)) \quad ((\forall x. A \ x) \lor B) \\
\text{eq} & \quad (A \ \text{imp} \ B) \quad ((\neg A) \lor B) \\
\text{eq} & \quad (\neg (A \ \text{and} \ B)) \quad ((\neg A) \lor (\neg B))
\end{align*}
\]

How can we efficiently store and retrieve data?

- Share term structure
- Share common operations
Indexing

Set of terms

\[
\begin{align*}
\text{eq} & \quad (\forall x. (A \, x) \lor B) \quad (\forall x. A \, x) \lor B \\
\text{eq} & \quad (A \rightarrow B) \quad (\neg A) \lor B \\
\text{eq} & \quad (\neg (A \land B)) \quad (\neg A) \lor (\neg B)
\end{align*}
\]

How can we efficiently store and retrieve data?

- Share term structure
- Share common operations
- Even below a binder!

\[
\begin{align*}
\text{eq} & \quad (\forall x. (A \, x) \rightarrow B) \quad (\exists x. A \, x) \rightarrow B \\
\text{eq} & \quad (\forall x. (A \, x) \lor B) \quad (\forall x. A \, x) \lor B
\end{align*}
\]
Set of terms

\[
\begin{align*}
\text{eq} (\forall x. ((A \ x) \lor B)) & \quad \text{((all } \lambda x. \ A \ x \text{) or } B) \\
\text{eq} (A \ \text{imp} \ B) & \quad \text{((not } A \text{) or } B) \\
\text{eq} (\text{not } (A \ \text{and} \ B)) & \quad \text{((not } A \text{) or (not } B))
\end{align*}
\]

How can we efficiently store and retrieve data?

- Share term structure
- Share common operations
- Even below a binder!

\[
\begin{align*}
\text{eq} (\forall x. (A \ x) \ \text{imp} \ B) & \quad \text{((exists } \lambda x. \ A \ x \text{) imp } B) \\
\text{eq} (\forall x. (A \ x) \ or \ B) & \quad \text{((all } \lambda x. \ A \ x \text{) or } B)
\end{align*}
\]
Step 1: Linearization

Set of linear terms

1. \( \text{eq } (\forall x. ((A \land x) \lor (B' \land x))) \quad (\forall x. (A' \land x) \lor B) \)
2. \( \text{eq } (A \implies B) \quad ((\neg A') \lor B') \)
3. \( \text{eq } ((\neg (A \land B)) \quad ((\neg A') \lor (\neg B')) \)

Constraints

\( A = A', \quad \forall x. B' \land x \\
A' \equiv A, \quad B \\
A' \equiv A, \quad B \equiv B' \)

- Linearize every terms
  Factor out “hard” sub-expressions
- Uniform naming for variables
Step 2: Common sub-expression

Set of linear terms

(1) eq (all λ x. ((A x) or (B' x))) ((all λ x. A' x) or B)
(2) eq (A imp B) ((not A') or B')
(3) eq (not (A and B)) ((not A') or (not B'))

Constraints

∀ x. B' x \equiv B, \quad A = A'
A' \equiv A, \quad B \equiv B'
A' \equiv A, \quad B \equiv B'

• Factor out common sub-expressions!

  eq (A imp B) ((not A') or B')
  eq (not (A and B)) ((not A') or (not B'))
  eq i_1 ((not A') or i_2)
Step 2: Common sub-expression

Set of linear terms

(1) eq (all λ x. ((A x) or (B' x))) ((all λ x. A' x) or B)
(2) eq (A imp B) ((not A') or B')
(3) eq (not (A and B)) ((not A') or (not B'))

Constraints

∀ x. B' x ⊨ B, A = A'
A' ⊨ A, B ⊨ B'
A' ⊨ A, B ⊨ B'

• Factor out common sub-expressions!
  
  eq (bold A imp B) ((not A') or B')
  eq (not (bold A and bold B)) ((not A') or (not B'))
  
  eq i_1 ((not A') or i_2)

• In general the most specific common generalization does not exist!

Key: linearization
Higher-order substitution trees

Set of linear terms

(1) eq (all λ x. ((A x) or (B’ x))) ((all λ x. A’ x) or B)
(2) eq (A imp B) ((not A’) or B’)
(3) eq (not (A and B)) ((not A’) or (not B’))

Compose substitutions!

Overcoming Performance Barriers: – p.30/38
## Parser for formulas

<table>
<thead>
<tr>
<th>#tok</th>
<th>iterative deepening</th>
<th>memoization noindex</th>
<th>memoization index</th>
<th>reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.98 sec</td>
<td>0.13 sec</td>
<td>0.07 sec</td>
<td>46%</td>
</tr>
<tr>
<td>58</td>
<td>∞</td>
<td>2.61 sec</td>
<td>1.25 sec</td>
<td>52%</td>
</tr>
<tr>
<td>117</td>
<td>∞</td>
<td>10.44 sec</td>
<td>5.12 sec</td>
<td>51%</td>
</tr>
<tr>
<td>235</td>
<td>∞</td>
<td>75.57 sec</td>
<td>26.08 sec</td>
<td>66%</td>
</tr>
</tbody>
</table>

∞ = process does not terminate in 6h

Intel Pentium 1.6GHz, RAM 256MB, SML New Jersey 110, Twelf 1.4.
## Refinement type-checking

<table>
<thead>
<tr>
<th>example</th>
<th>noindex</th>
<th>index</th>
<th>reduction</th>
<th>orig</th>
</tr>
</thead>
<tbody>
<tr>
<td>First sub</td>
<td>3.19 sec</td>
<td>0.46 sec</td>
<td>86%</td>
<td></td>
</tr>
<tr>
<td>answer mult</td>
<td>7.78 sec</td>
<td>0.89 sec</td>
<td>89%</td>
<td></td>
</tr>
<tr>
<td>square</td>
<td>9.02 sec</td>
<td>0.98 sec</td>
<td>89%</td>
<td></td>
</tr>
<tr>
<td>Not mult</td>
<td>2.38 sec</td>
<td>0.38 sec</td>
<td>84%</td>
<td></td>
</tr>
<tr>
<td>proveable plus</td>
<td>6.48 sec</td>
<td>0.85 sec</td>
<td>87%</td>
<td></td>
</tr>
<tr>
<td>square</td>
<td>9.29 sec</td>
<td>1.09 sec</td>
<td>88%</td>
<td></td>
</tr>
<tr>
<td>All sub</td>
<td>6.88 sec</td>
<td>0.71 sec</td>
<td>90%</td>
<td></td>
</tr>
<tr>
<td>answers mult</td>
<td>9.06 sec</td>
<td>0.98 sec</td>
<td>89%</td>
<td></td>
</tr>
<tr>
<td>square</td>
<td>10.30 sec</td>
<td>1.08 sec</td>
<td>90%</td>
<td></td>
</tr>
</tbody>
</table>
## Refinement type-checking

<table>
<thead>
<tr>
<th>example</th>
<th>noindex</th>
<th>index</th>
<th>time red.</th>
<th>orig</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>sub</td>
<td>3.19 sec</td>
<td>0.46 sec</td>
<td>86%</td>
</tr>
<tr>
<td>answer</td>
<td>mult</td>
<td>7.78 sec</td>
<td>0.89 sec</td>
<td>89%</td>
</tr>
<tr>
<td></td>
<td>square</td>
<td>9.02 sec</td>
<td>0.98 sec</td>
<td>89%</td>
</tr>
<tr>
<td>Not</td>
<td>mult</td>
<td>2.38 sec</td>
<td>0.38 sec</td>
<td>84%</td>
</tr>
<tr>
<td>provable</td>
<td>plus</td>
<td>6.48 sec</td>
<td>0.85 sec</td>
<td>87%</td>
</tr>
<tr>
<td></td>
<td>square</td>
<td>9.29 sec</td>
<td>1.09 sec</td>
<td>88%</td>
</tr>
<tr>
<td>All</td>
<td>sub</td>
<td>6.88 sec</td>
<td>0.71 sec</td>
<td>90%</td>
</tr>
<tr>
<td>answers</td>
<td>mult</td>
<td>9.06 sec</td>
<td>0.98 sec</td>
<td>89%</td>
</tr>
<tr>
<td></td>
<td>square</td>
<td>10.30 sec</td>
<td>1.08 sec</td>
<td>90%</td>
</tr>
</tbody>
</table>
Contribution and related work

- Contribution:
  - Higher-order term indexing (key: linearization, $\eta$-longform)
  - Indexing substantially improves performance
    runtime reduced between 46% and 90% (ICLP’03)
  - Application: Small proof witness [ICLP’05]
  - Application: Propositional theorem proving [CADE’05]
Contribution and related work

• Contribution:
  – Higher-order term indexing (key: linearization, $\eta$-longform)
  – Indexing substantially improves performance
    runtime reduced between 46% and 90% (ICLP’03)
  – Application: Small proof witness [ICLP’05]
  – Application: Propositional theorem proving [CADE’05]

• Related Work:
  – Substitution trees for first-order terms [Graf95]
  – (Higher-order) automata-driven indexing [Necula,Rahul01]
    imperfect filter, full higher-order unification to check candidates
Outline

• Logical frameworks and applications
• Efficient proof search in logical frameworks
  - Optimizing higher-order unification
  - Higher-order term indexing
• Conclusion and future work
Conclusion

• This is opens many new opportunities
  – to experiment and develop large-scale systems.
    for example: proof-carrying code, POPLmark
  – to explore the full potential of logical frameworks
    new applications: authentication, security

• Efficient proof search techniques are critical
  – to sustain performance.
  – to reduce response time to the developer.
Future work

Narrowing the performance gap further

- Mode, determinism, termination analysis
  [Schrijvers et al. 02]
- Exploiting properties of local theories
  (joint work with Xi Li (McGill))

Tabled higher-order logic programming [Pie’03, Pie’05]

- Strongly connected components (SCC) [Swift, Sagonas98]
- Model-checking over high-level specifications
  [Ramakrishnan’97]
Finally ...

if you want to find out more:

http://www.cs.mcgill.ca/~bpientka

email: bpientka@cs.mcgill.ca