



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:
 λ Prolog[Nadathur'99], Twelf[Pf'99], Isabelle[Paulson86]
- Other higher-order systems: Coq, PVS, Nuprl, HOL, ...

Higher-order logic programming

- Higher-order data-types: dependently typed λ -calculus

Higher-order logic programming

- Higher-order data-types: dependently typed λ -calculus
- Dynamic program clauses: Clauses may contain nested universal quantifiers and implications

Higher-order logic programming

- Higher-order data-types: dependently typed λ -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 λ -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 λ -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

$$A ::= P \mid A \supset A \mid A \vee A \mid \neg A \mid \forall x.A \mid \exists x.A \mid \dots$$

- Specifying equivalence preserving transformations
- Sample rules:

$$A \supset B \quad \leftrightarrow \quad \neg A \vee B$$

$$\forall x.(A(x) \vee B) \quad \leftrightarrow \quad (\forall x.A(x)) \vee B$$

$$\forall x.(A(x) \supset B) \quad \leftrightarrow \quad (\exists x.A(x)) \supset B$$

if x is not free in B

Specification in LF

- 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:

$$A \supset B \iff \neg A \vee B$$

$eq_imp: eq \ (A \ imp \ B) \quad ((not \ A) \ or \ B)$

Specification in LF

- 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:

$$\forall x.(A(x) \supset B) \iff (\exists x.A(x)) \supset B$$

$eq_all: eq \ (all \ (\lambda x. (A \ x) \ imp \ B)) \quad ((exists \ (\lambda x. A \ x)) \ imp \ B).$

Specification in LF

- 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.

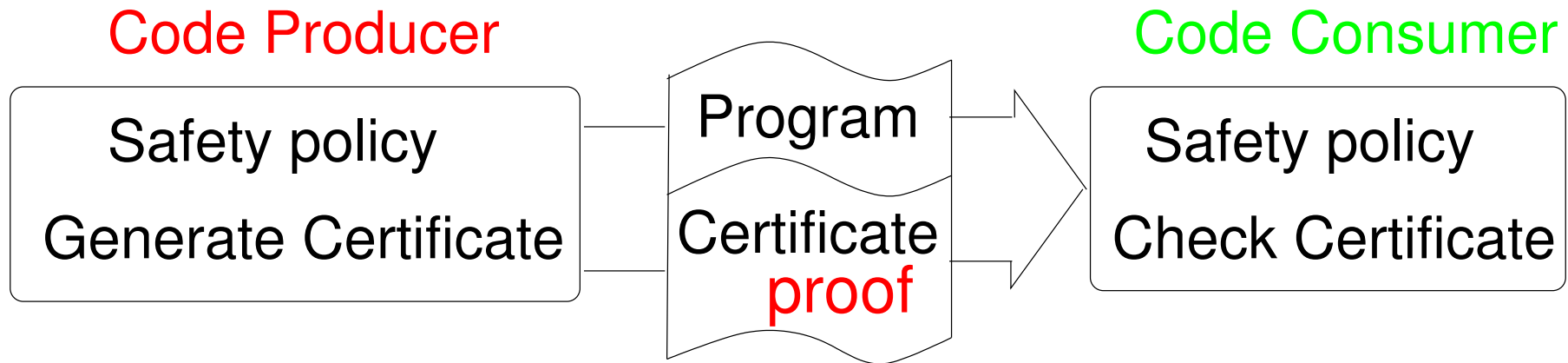
- Transforming propositions:

$$\forall x.(A(x) \supset B) \iff (\exists x.A(x)) \supset B$$

eq_all: eq (all ($\lambda x.$ (A x) imp B)) ((exists ($\lambda x.$ A x)) imp B).

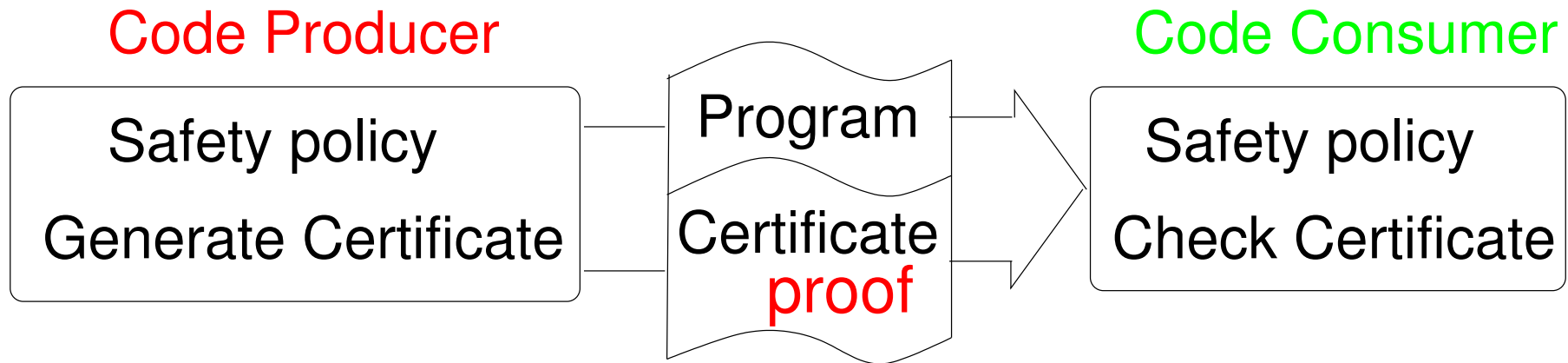
- A: i \rightarrow o and B: o are **meta-variables**
also sometimes called *existential variables* or *logic variables*

Application: certified code



- Foundational proof-carrying code : [Appel, Felty 00]
- Temporal-logic proof carrying code [Bernard, Lee02]
- Foundational typed assembly language : [Crary 03]
- Distributed access control: [Bauer, Reiter'05]

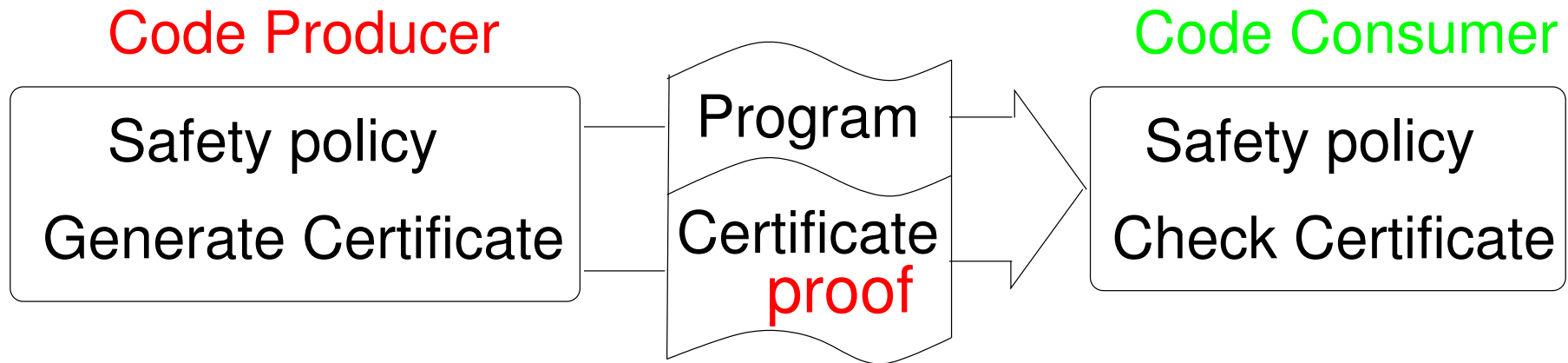
Application: certified code



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, Lee98] [Reed'04]
- Proof checking via “higher-order” logic programming [Necula'01], [Wu'03]

Application: Verified Software

- 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.
 - ...

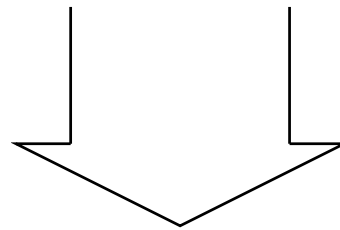
State of the art

- 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

Problem 1

“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]

Problem 1

“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!

Problem 1

“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:

eq $(A \text{ imp } B)$ $((\text{not } A) \text{ or } B)$ Success

eq $(p \text{ imp } q)$ $((\text{not } C) \text{ or } q)$ $A = p, B = q, C=A$

Basic operation: unification

- Example 1:

eq (A imp B) ((not A) or B) Success

eq (p imp q) ((not C) or q) A = p, B = q, C=A

- Example 2:

eq (A imp B) ((not A) or B) Failure(occurs-check!)
C = (A imp B),
eq C ((not C) or q) A = C, B = q

Basic operation: unification

- Example 1:

eq (A imp B) ((not A) or B) Success

eq (p imp q) ((not C) or q) A = p, B = q, C=A

- Example 2:

eq (A imp B) ((not A) or B) Failure(occurs-check!)
C = (A imp B),
eq C ((not C) or q) A = C, B = q

- Occurs check is expensive!

Basic operation: unification

- Example 1:

eq (A imp B) ((not A) or B) Success

eq (p imp q) ((not C) or q) A = p, B = q, C=A

- Example 2:

eq (A imp B) ((not A) or B) Failure(occurs-check!)
C = (A imp B),
eq C ((not C) or q) A = C, B = q

- 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

(all $\lambda x. ((A\ x)\ \text{imp}\ B))$ – ok

$((C\ T)\ \text{imp}\ B)$ – not ok!

Higher-order pattern unification

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

(all $\lambda x. ((A x) \text{ imp } B))$ – ok

((C T) imp B) – not ok!

- Closed instantiation for meta-variables!

eq (all $\lambda y. \underline{((p y) \text{ imp } (p y))}$ imp q) C
=

eq (all $\lambda x. \underline{(A x)}$ imp B) ((exists $\lambda x. A x$) imp B)

Higher-order pattern unification

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

$(\text{all } \lambda x. ((A x) \text{ imp } B))$ – ok

$((C T) \text{ imp } B)$ – not ok!

- Closed instantiation for meta-variables!

eq $(\text{all } \lambda y. \underline{((p y) \text{ imp } (p y))} \text{ imp } \underline{q}) \quad \underline{C}$

=

eq $(\text{all } \lambda x. \underline{(A x)} \text{ imp } \underline{B}) \quad \underline{((\text{exists } \lambda x. A x) \text{ imp } B)}$

- Solution:
A = $\lambda z. (p z) \text{ imp } (p z)$
B = q
C = $((\text{exists } (\lambda x. A x)) \text{ imp } B)$
= $(\text{imp } (\text{exists } (\lambda x. \text{imp } (p x) (p x)))) q$

Higher-order pattern unification

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

(all $\lambda x. ((A x) \text{ imp } B))$ – ok

((C T) imp B) – not ok!

- Closed instantiation for meta-variables?

eq (all $\lambda y. \underline{((p y) \text{ imp } (p y))} \text{ imp } \underline{(p y)} \underline{C}$)
=

eq (all $\lambda x. \underline{(A x)} \text{ imp } \underline{B} \underline{((exists \lambda x. A x) \text{ imp } B)}$)

Higher-order pattern unification

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

(all $\lambda x.$ ((**A** x) imp **B**)) – ok

((**C** **T**) imp **B**) – not ok!

- Closed instantiation for meta-variables?

eq (all $\lambda y.$ ((p y) imp (p y)) imp (p y) **C**
=

eq (all $\lambda x.$ (**A** x) imp **B** ((exists $\lambda x.$ **A** x) imp **B**)

- Failure $A = \lambda z. (p z) \text{ imp } (p z)$

$B = ?$ There is no closed instantiation for B !

$C = \dots$

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!

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:

$$\text{eq } (A \text{ imp } B) \quad ((\text{not } A) \text{ or } B)$$
$$\iff$$
$$\text{eq } (A \text{ imp } B) \quad ((\text{not } A') \text{ or } B') \quad \text{and } A' \doteq A \text{ and } B' \doteq B$$

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:

eq (all $\lambda x. (A\ x) \text{ imp } B$) ((exists $\lambda x. (A\ x)$) imp B)

\iff

eq (all $\lambda x. (A\ x) \text{ imp } (B'\ x)$) ((exists $\lambda x. A'\ x$) imp B)

$A' \doteq A$ and $\forall x. (B'\ x) \doteq B$

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

example	standard	opt	reduction
mul2	9.52 sec	5.51 sec	42.86%
div2	153.61 sec	121.96 sec	20.63%
pack	1075.61 sec	197.07 sec	81.65%
divx	1133.15 sec	333.69 sec	70.50%
listsum	∞	1073.33 sec	100%

∞ = 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: λ Prolog (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: λ -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

Outline

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

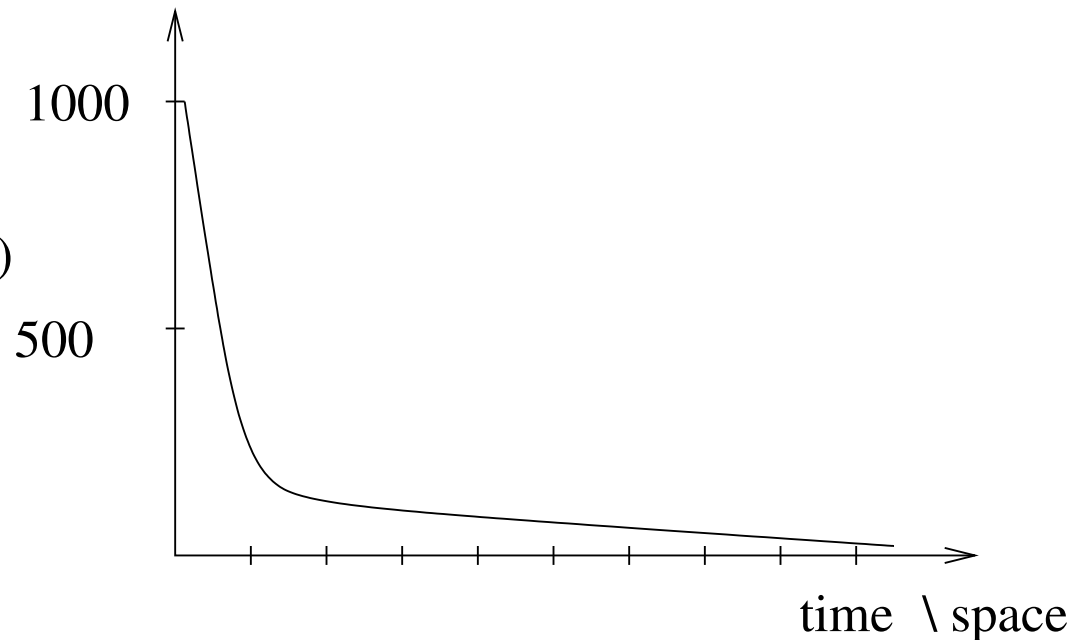
Problem 2

“...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]

Problem 2

“...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]

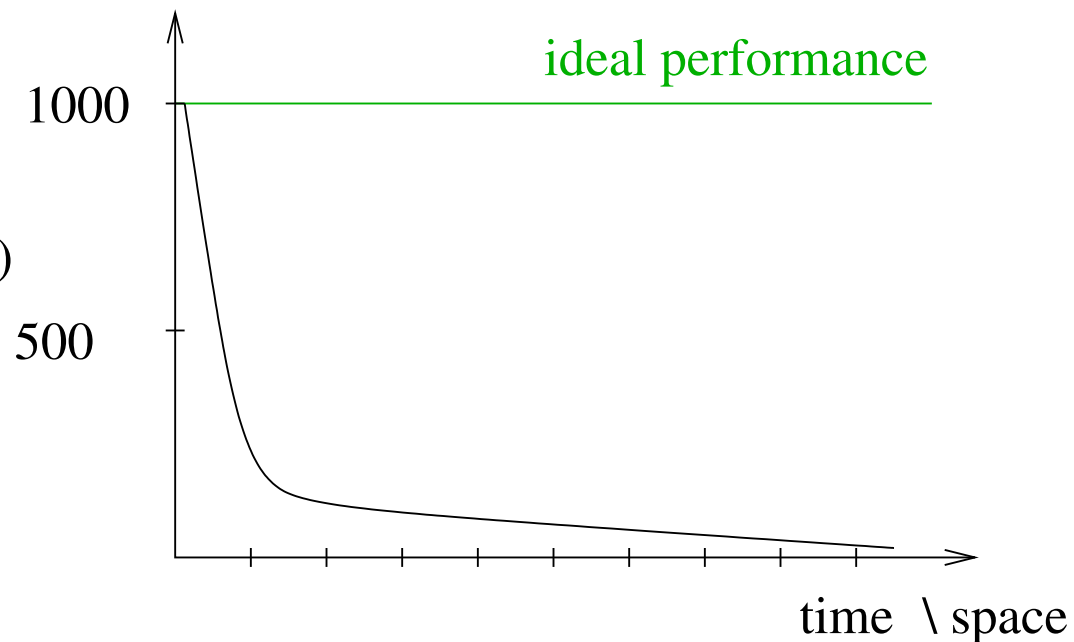
number of
conclusions (= proof steps)



Problem 2

“...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]

number of
conclusions (= proof steps)



Indexing

Set of terms

eq (all $\lambda x. ((A x) \text{ or } B)$) ((all $\lambda x. A x$) or B)

eq (A imp B) ((not A) or B)

eq (not (A and B)) ((not A) or (not B))

How can we efficiently store and retrieve data?

Indexing

Set of terms

eq (all $\lambda x. ((A x) \text{ or } B))$ $((\text{all } \lambda x. A x) \text{ or } B)$

eq $(A \text{ imp } B)$ $((\text{not } A) \text{ or } B)$

eq $(\text{not } (A \text{ and } B))$ $((\text{not } A) \text{ or } (\text{not } B))$

How can we efficiently store and retrieve data?

- Share term structure
- Share common operations

Indexing

Set of terms

eq (all $\lambda x. ((A\ x) \text{ or } B))$ ((all $\lambda x. A\ x$) or B)

eq (A imp B) ((not A) or B)

eq (not (A and B)) ((not A) or (not B))

How can we efficiently store and retrieve data?

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

eq (all $\lambda x. (A\ x) \text{ imp } B$) ((exists $\lambda x. A\ x$) imp B)

eq (all $\lambda x. (A\ x) \text{ or } B$) ((all $\lambda x. A\ x$) or B)

Indexing

Set of terms

eq $(\text{all } \lambda x. ((A x) \text{ or } B))$ $((\text{all } \lambda x. A x) \text{ or } B)$

eq $(A \text{ imp } B)$ $((\text{not } A) \text{ or } B)$

eq $(\text{not } (A \text{ and } B))$ $((\text{not } A) \text{ or } (\text{not } B))$

How can we efficiently store and retrieve data?

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

eq $(\text{all } \lambda x. (A x) \text{ imp } B)$ $((\text{exists } \lambda x. A x) \text{ imp } B)$

eq $(\text{all } \lambda x. (A x) \text{ or } B)$ $((\text{all } \lambda x. A x) \text{ or } B)$

Step 1: Linearization

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

- $A = A', \quad \forall x. B' x \doteq B$
- $A' \doteq A, \quad B \doteq B'$
- $A' \doteq A, \quad B \doteq B'$

- Linearize every terms
Factor out “hard” sub-expressions
- Uniform naming for variables

Step 2: Common sub-expression

Set of linear terms

Constraints

(1) eq (all λ x. ((A x) or (B' x)))	((all λ x. A' x) or B)	$\forall x. B' x \doteq B, A = A'$
(2) eq (A imp B)	((not A') or B')	$A' \doteq A, B \doteq B'$
(3) eq (not (A and B))	((not A') or (not B'))	$A' \doteq A, B \doteq B'$

- 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

Constraints

(1) eq (all λ x. ((A x) or (B' x)))	((all λ x. A' x) or B)	$\forall x. B' x \doteq B, A = A'$
(2) eq (A imp B)	((not A') or B')	$A' \doteq A, B \doteq B'$
(3) eq (not (A and B))	((not A') or (not B'))	$A' \doteq A, B \doteq B'$

- 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)

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

Key: linearization

Parser for formulas

#tok	iterative	memoization		reduction
	deepening	noindex	index	
20	0.98 sec	0.13 sec	0.07 sec	46%
58	∞	2.61 sec	1.25 sec	52%
117	∞	10.44 sec	5.12 sec	51%
235	∞	75.57 sec	26.08 sec	66%

∞ = process does not terminate in 6h

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

Refinement type-checking

	example	noindex	index	reduction	orig
First	sub	3.19 sec	0.46 sec	86%	
answer	mult	7.78 sec	0.89 sec	89%	
	square	9.02 sec	0.98 sec	89%	
Not provable	mult	2.38 sec	0.38 sec	84%	
	plus	6.48 sec	0.85 sec	87%	
	square	9.29 sec	1.09 sec	88%	
All answers	sub	6.88 sec	0.71 sec	90%	
	mult	9.06 sec	0.98 sec	89%	
	square	10.30 sec	1.08 sec	90%	

Refinement type-checking

	example	noindex	index	time red.	orig
First answer	sub	3.19 sec	0.46 sec	86%	0.15 sec
	mult	7.78 sec	0.89 sec	89%	0.15 sec
	square	9.02 sec	0.98 sec	89%	0.16 sec
Not provable	mult	2.38 sec	0.38 sec	84%	13.50 sec
	plus	6.48 sec	0.85 sec	87%	∞
	square	9.29 sec	1.09 sec	88%	∞
All answers	sub	6.88 sec	0.71 sec	90%	5.59 sec
	mult	9.06 sec	0.98 sec	89%	∞
	square	10.30 sec	1.08 sec	90%	∞

Contribution and related work

- Contribution:
 - Higher-order term indexing (key: linearization, η -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, η -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 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/~bpienkka>

email: bpienkka@cs.mcgill.ca