Tabled higher-order logic programming

Brigitte Pientka

Department of Computer Science

Carnegie Mellon University

Pittsburgh, PA, 15213-3891, USA

Thesis Committee: Frank Pfenning (Chair)

Robert Harper

Dana Scott

David Warren, University of New York at Stony Brook

Outline

- Logical frameworks and certified code
- Tabled higher-order logic programming
  - Basic idea and challenges
  - Experiments and Evaluation
  - Improving efficiency
- Conclusion and future work

Deductive systems and logical frameworks

Deductive systems are plentiful computer science.

- Axioms and inference rules
- Examples: operational semantics, type system, logic, etc.

Deductive systems and logical frameworks

Deductive systems are plentiful computer science.

- Axioms and inference rules
- Examples: operational semantics, type system, logic, etc.

Logical framework: meta-language for deductive systems

- High-level specifications (e.g. type system)
- Execution via logic programming interpretation (e.g. type checker)
- Meta-reasoning via theorem prover combining induction and logic programming search (e.g. type preservation)

Declarative description of subtyping

types τ ::= zero | pos | nat | bit | $\tau_1 \Rightarrow \tau_2$ | $\ldots$

Example: $6 = \epsilon 110$ and $\epsilon 110 \in$ nat

Declarative description of subtyping

types τ ::= zero | pos | nat | bit | $\tau_1 \Rightarrow \tau_2$ | ...

Example: $6 = \epsilon 110$ and $\epsilon 110 \in$ nat

$$\frac{}{\text{zero} \preceq \text{nat}}\ \text{zn} \qquad \frac{}{\text{pos} \preceq \text{nat}}\ \text{pn} \qquad \frac{}{\text{nat} \preceq \text{bit}}\ \text{nb}$$

Declarative description of subtyping

types τ ::= zero | pos | nat | bit | $\tau_1 \Rightarrow \tau_2$ | $\ldots$

Example: $6 = \epsilon 110$ and $\epsilon 110 \in$ nat

$$\frac{}{\text{zero} \preceq \text{nat}}\ \text{zn} \qquad \frac{}{\text{pos} \preceq \text{nat}}\ \text{pn} \qquad \frac{}{\text{nat} \preceq \text{bit}}\ \text{nb}$$

$$\frac{}{\tau \preceq \tau}\ \text{refl} \qquad \frac{\tau_1 \preceq \tau \qquad \tau \preceq \tau_2}{\tau_1 \preceq \tau_2}\ \text{tr}$$

Typing rules for Mini-ML

expressions e ::= $\epsilon \mid e\,0 \mid e\,1 \mid$ fun $x.e \mid$ app $e_1\ e_2$

$$\frac{\Gamma \vdash e : \tau' \qquad \tau' \preceq \tau}{\Gamma \vdash e : \tau}\ \text{tp-sub}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{fun}\ x.e : \tau_1 \Rightarrow \tau_2}\ \text{tp-fun}$$

Implementation of subtyping

```
zn:  sub zero nat.
pn:  sub pos nat.
nb:  sub nat bit.
refl:  sub T T.
tr:  sub T S
     <- sub T R
     <- sub R S.
```

Implementation of subtyping

```
zn:  sub zero nat.
pn:  sub pos nat.        ?- sub zero bit.
nb:  sub nat bit.
refl:  sub T T.
tr:  sub T S
     <- sub T R
     <- sub R S.
```

Implementation of subtyping

```
zn:   sub zero nat.
pn:   sub pos nat.
nb:   sub nat bit.
refl:  sub T T.
tr:   sub T S
      <- sub T R
      <- sub R S.
```

```
?- sub zero bit.
```

yes

Proof: `(tr nb zn)`

Implementation of typing rules

```
tp_sub:  of E T
      <- of E T'
      <- sub T' T.

tp_fun:  of (fun λx.E x) (T1 => T2)
      <-(Πx:exp.of x T1 -> of (E x) T2).
```

"forall `x:exp`, assume `of x T1`
and show `of (E x) T2`"

Higher-order logic programming

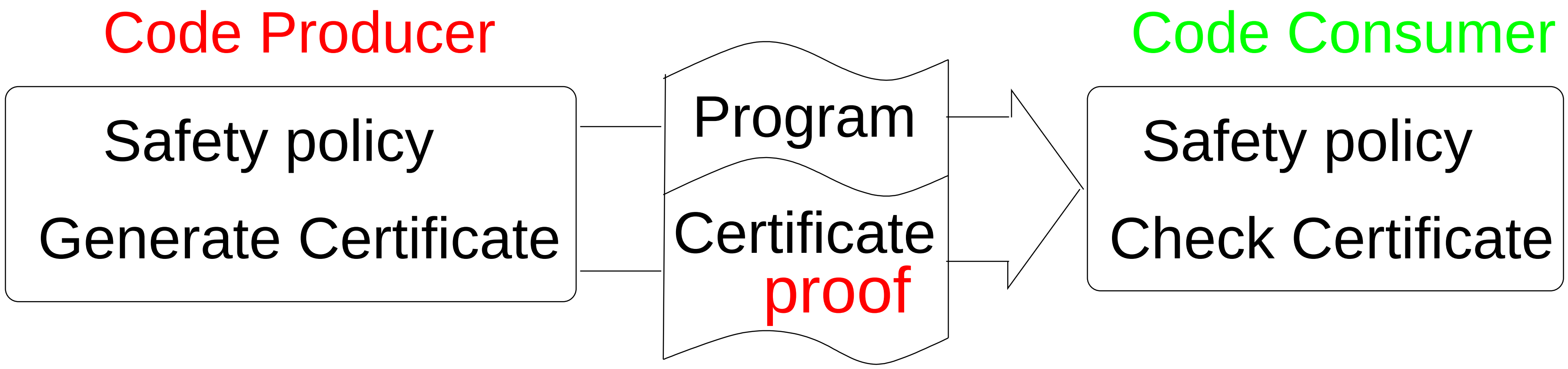
- Higher-order data-types:
  - λ-abstraction
  - dependent types
- Dynamic program clauses
- Explicit proof objects

Higher-order logic programming

- Higher-order data-types:
  - λ-abstraction
  - dependent types
- Dynamic program clauses
- Explicit proof objects

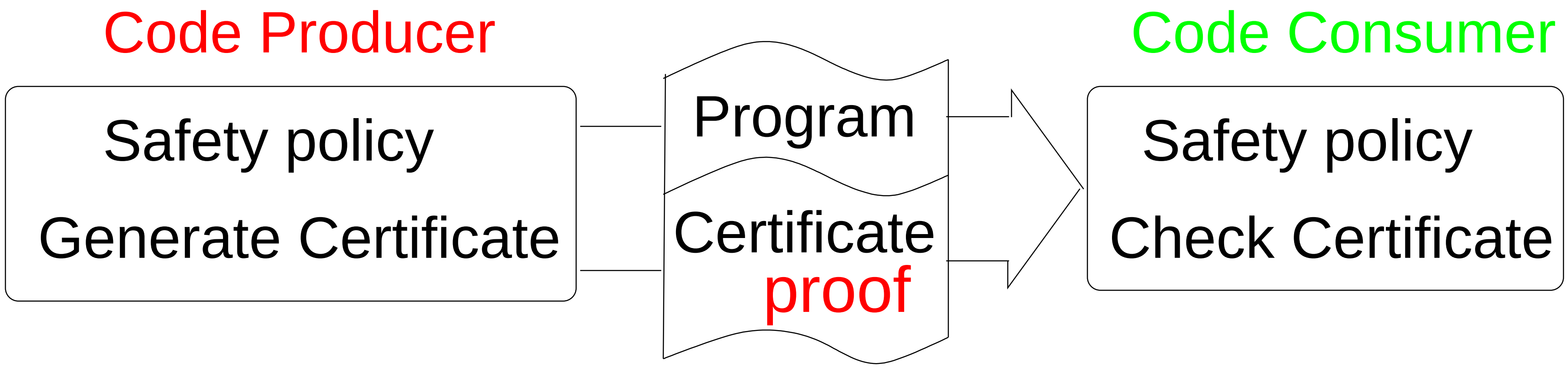
Different approaches: λProlog, Isabelle, Twelf

Application: certified code



- Foundational proof-carrying code : [Appel, Felty 00]
- Temporal-logic proof carrying code [Bernard,Lee02]
- Foundational typed assembly language : [Crary 03]
- Proof-carrying authentication: [Felten, Appel 99]

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

Some limitations in practice

- Straightforward specifications are not executable.
- Redundancy severely hampers performance.
- Meta-reasoning capabilities limited in practice.

Overcome some of these limitations using
tabelling and other optimizations!

This thesis

Tabled higher-order logic programming allows us to

- efficiently execute logical systems
  (interpreter using tabled search)
- automate the reasoning with and about them.
  (meta-theorem prover using tabled search)

This is a significant step towards
applying logical frameworks in practice.

Contributions

Tabled higher-order logic programming

- Characterization based on uniform proofs (ICLP'02)
- Implementation of a tabled interpreter
- Case studies (parsing, refinement types, rewriting)(LFM'02)

Efficient data-structures and algorithms

- Foundation for meta-variables (LFM'03)
- Optimizing higher-order unification (CADE'03)
- Higher-order term indexing (ICLP'03)

Meta-reasoning based on tabled search

Outline

- Logical frameworks and certified code
- Tabled higher-order logic programming
  - Basic idea and challenges
  - Experiments and Evaluation
  - Improving efficiency
- Conclusion and future work

Outline

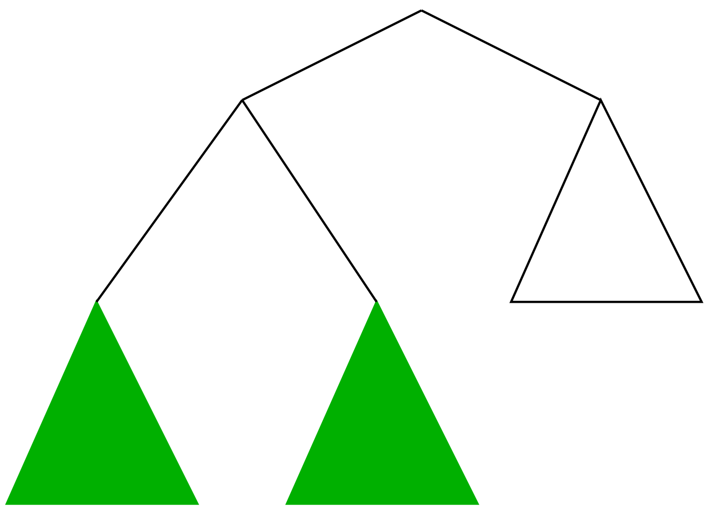
- Logical frameworks and certified code
- Tabled higher-order logic programming
  - Basic idea and challenges
  - Experiments and Evaluation
  - Improving efficiency
- Conclusion and future work

The idea

"...it is very common for the proofs to have repeated sub-proofs that should be hoisted out and proved only once ..." [Necula,Lee97]

The idea

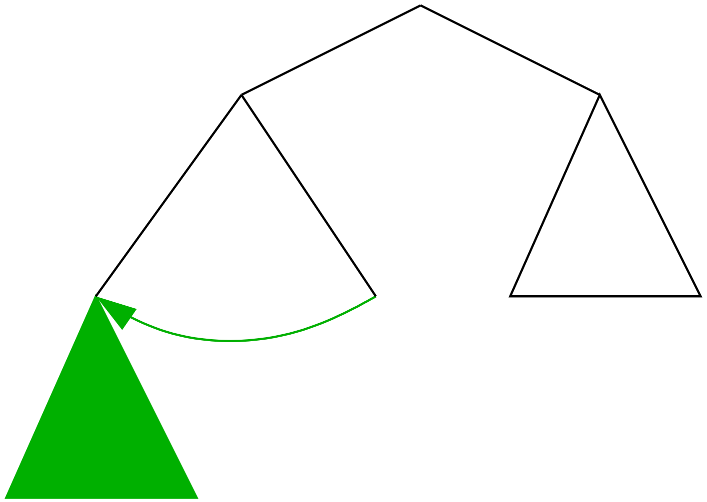
"...it is very common for the proofs to have repeated sub-proofs that should be hoisted out and proved only once ..." [Necula,Lee97]



Redundant computation

The idea

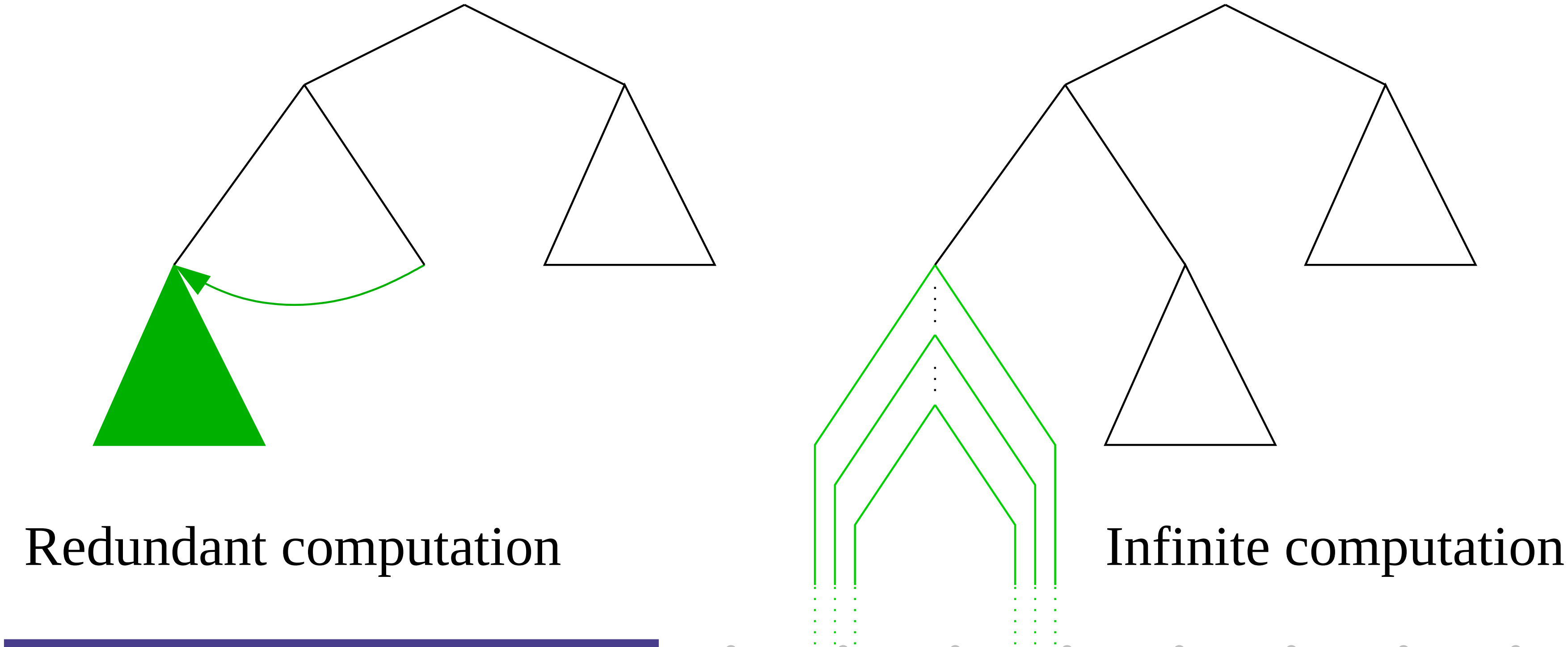
“...it is very common for the proofs to have repeated sub-proofs that should be hoisted out and proved only once ...” [Necula,Lee97]



Redundant computation

The idea

“...it is very common for the proofs to have repeated sub-proofs that should be hoisted out and proved only once ...” [Necula,Lee97]



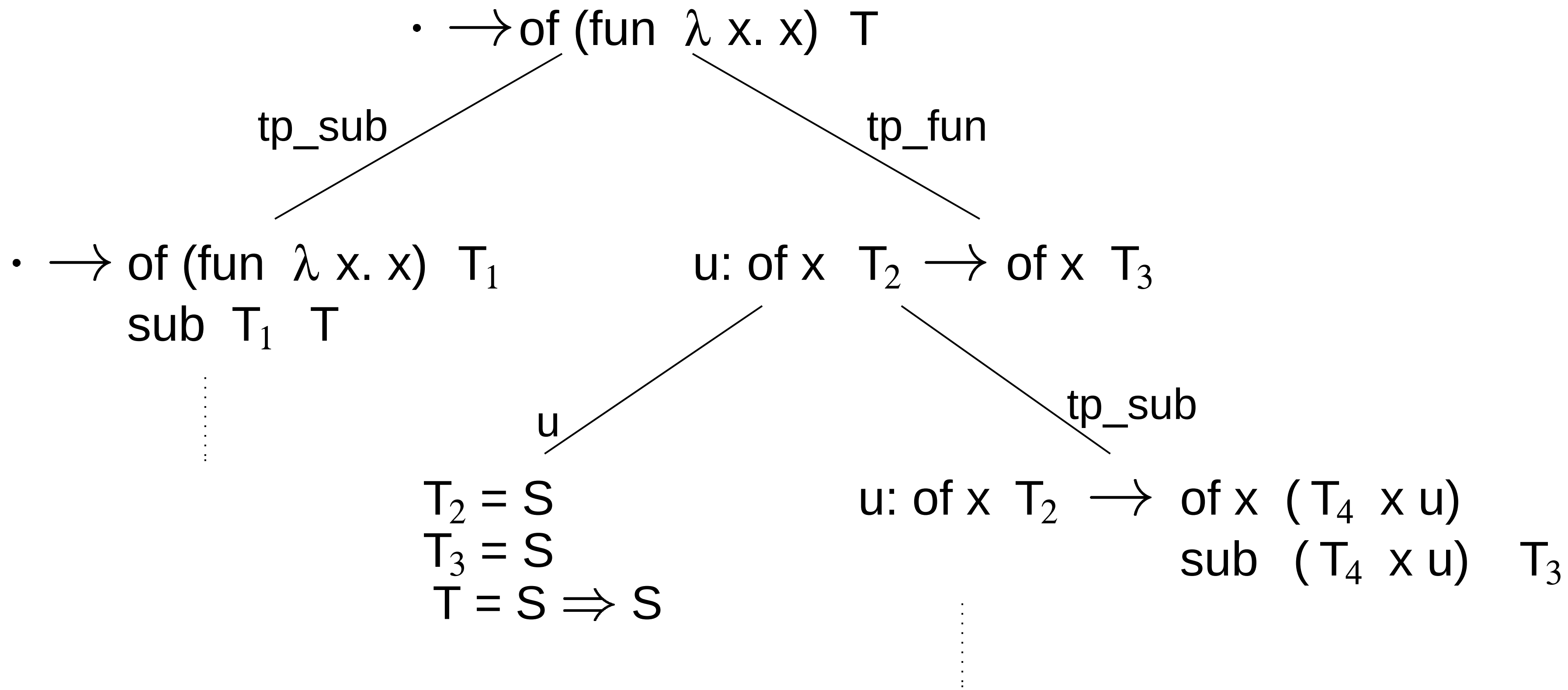
Recall...subtyping

```
tp_sub:  of E T
      <- of E T'
      <- sub T' T.

tp_fun:  of (fun λx.E x) (T1 => T2)
      <-(Πx:exp.of x T1 -> of (E x) T2).
```

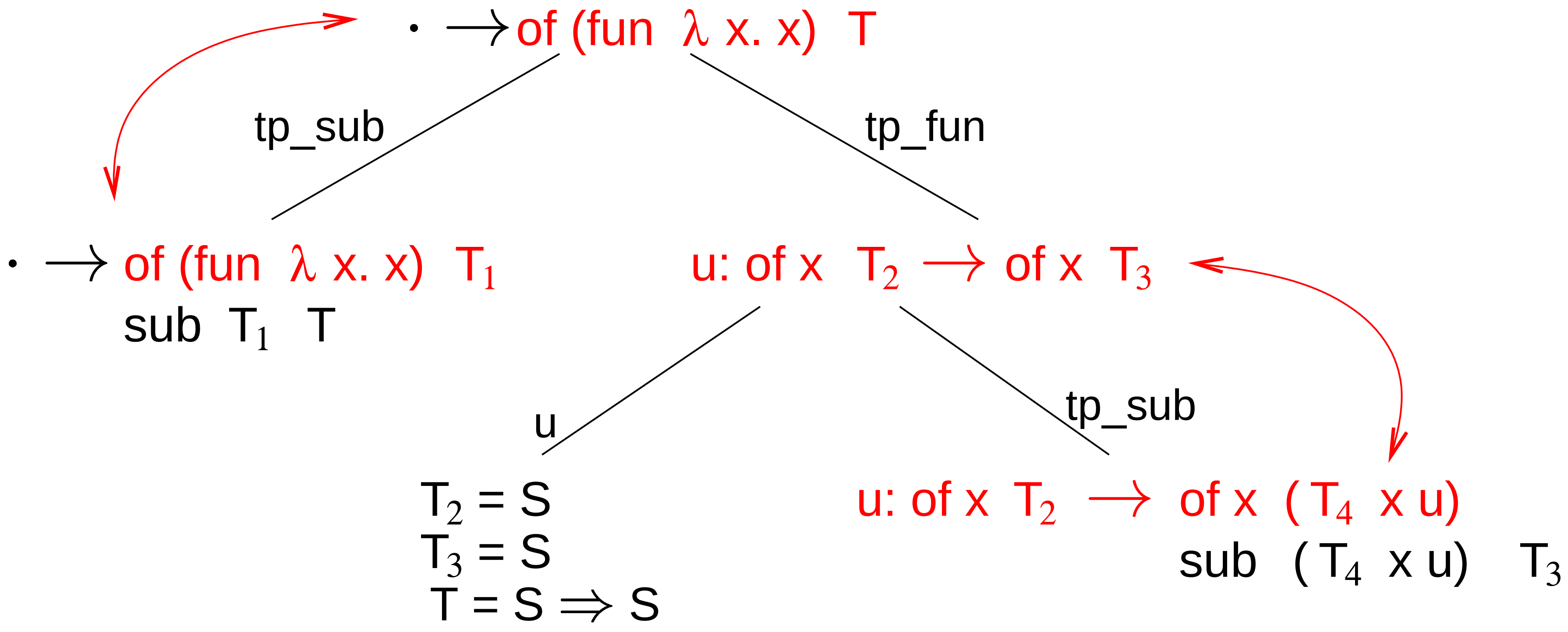
"forall `x:exp`, assume `of x T1`
and show `of (E x) T2`"

Proof tree



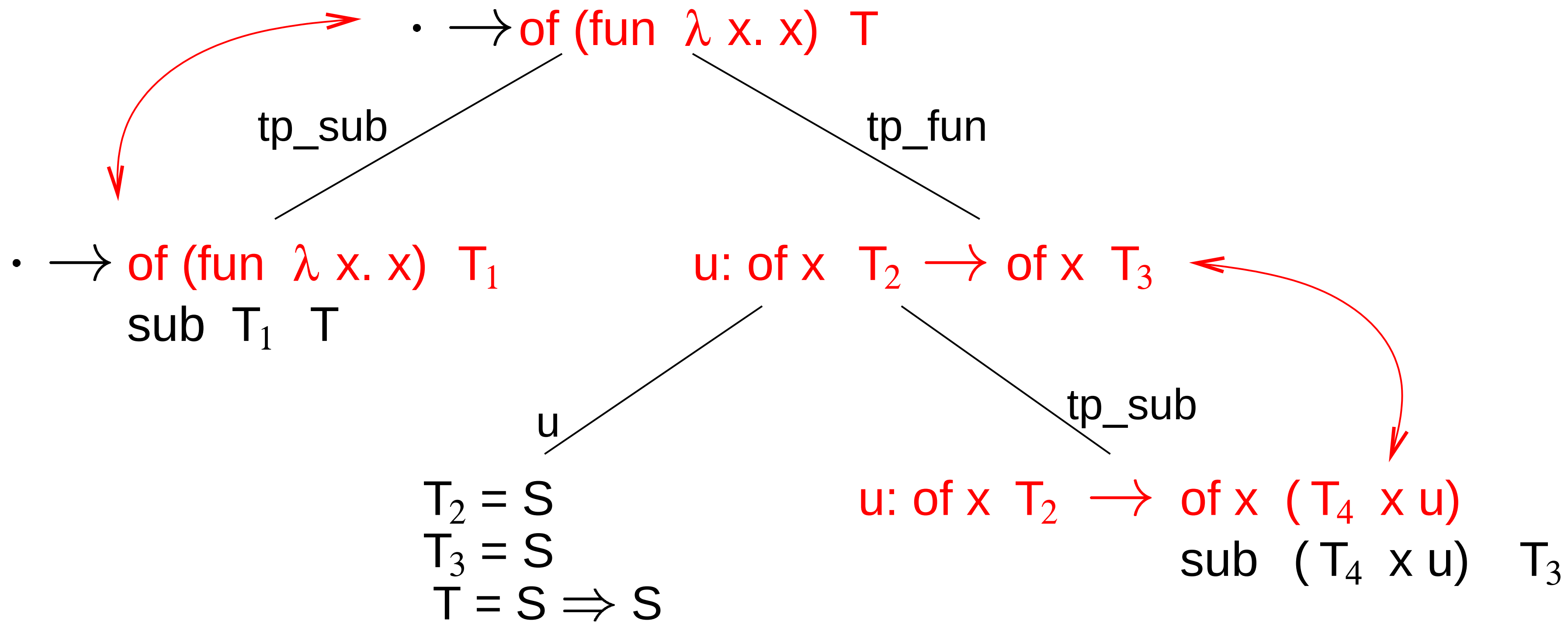
· ⟶ of (fun λ x. x) T
tp_sub
tp_fun
· ⟶ of (fun λ x. x) T_1
sub T_1 T
u: of x T_2 ⟶ of x T_3
u
tp_sub
T_2 = S
T_3 = S
T = S ⇒ S
u: of x T_2 ⟶ of x (T_4 x u)
sub (T_4 x u) T_3

Proof tree



Loop detection

Proof tree



Loop detection

How can we detect loops?

Loops modulo strengthening

- Dependencies among terms

$$\text{u:of x } T_2 \longrightarrow \text{of x } (T_4 \text{ x u})$$

Loops modulo strengthening

- Dependencies among terms

$$\text{u:of x } T_2 \rightarrow \text{of x } (T_4 \text{ x u})$$

strengthen $\quad \text{u:of x } T_2 \rightarrow \text{of x } T_4$

Loops modulo strengthening

- Dependencies among terms

  u:of x $T_2 \longrightarrow$ of x (T_4 x u)

  strengthen u:of x $T_2 \longrightarrow$ of x T_4

- Dependencies among propositions

  u:of x $T_2 \longrightarrow$ sub (T_4 x u) T_3

Loops modulo strengthening

- Dependencies among terms

$$\text{u:of x } T_2 \rightarrow \text{of x } (T_4 \text{ x u})$$

strengthen $\text{u:of x } T_2 \rightarrow \text{of x } T_4$

- Dependencies among propositions

$$\text{u:of x } T_2 \rightarrow \text{sub } (T_4 \text{ x u}) \; T_3$$

strengthen: $\cdot \rightarrow \text{sub } T_4 \; T_3$

Loops modulo strengthening

- Dependencies among terms

$$\text{u:of x } T_2 \to \text{of x } (T_4 \text{ x u})$$

strengthen $\quad \text{u:of x } T_2 \to \text{of x } T_4$

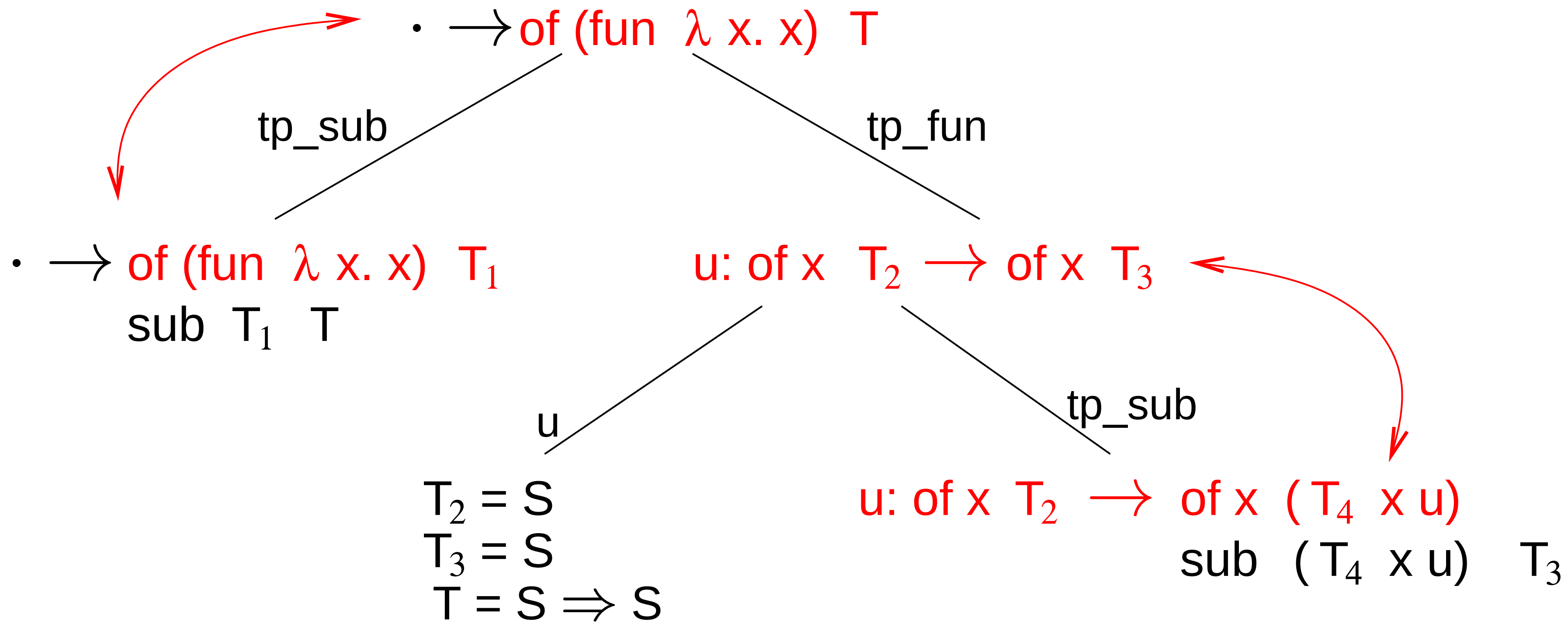
- Dependencies among propositions

$$\text{u:of x } T_2 \to \text{sub } (T_4 \text{ x u}) \; T_3$$

strengthen: $\quad \cdot \to \text{sub } T_4 \; T_3$

- Subordination analysis [Virga99]

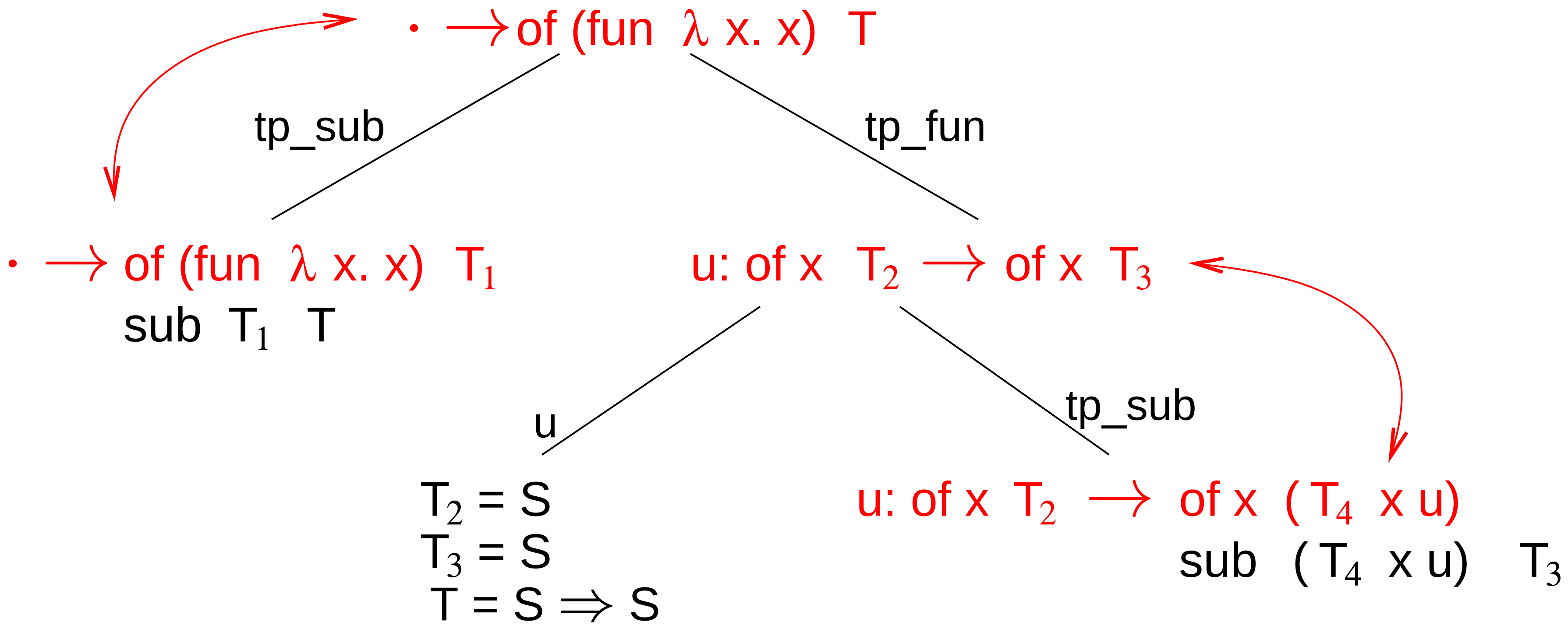
Proof tree (cont.)



Loop detection

How can we detect loops?

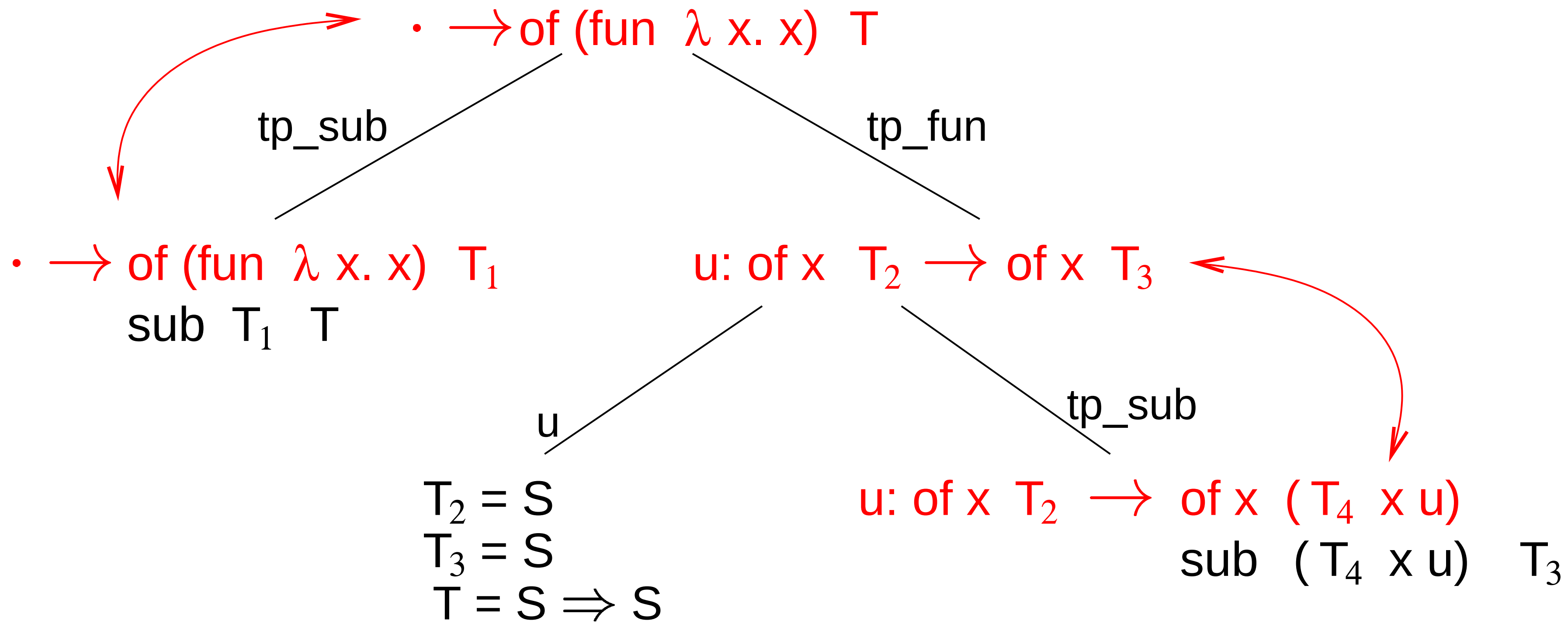
Proof tree (cont.)



Loop detection

How can we detect loops? Subordination

Proof tree (cont.)

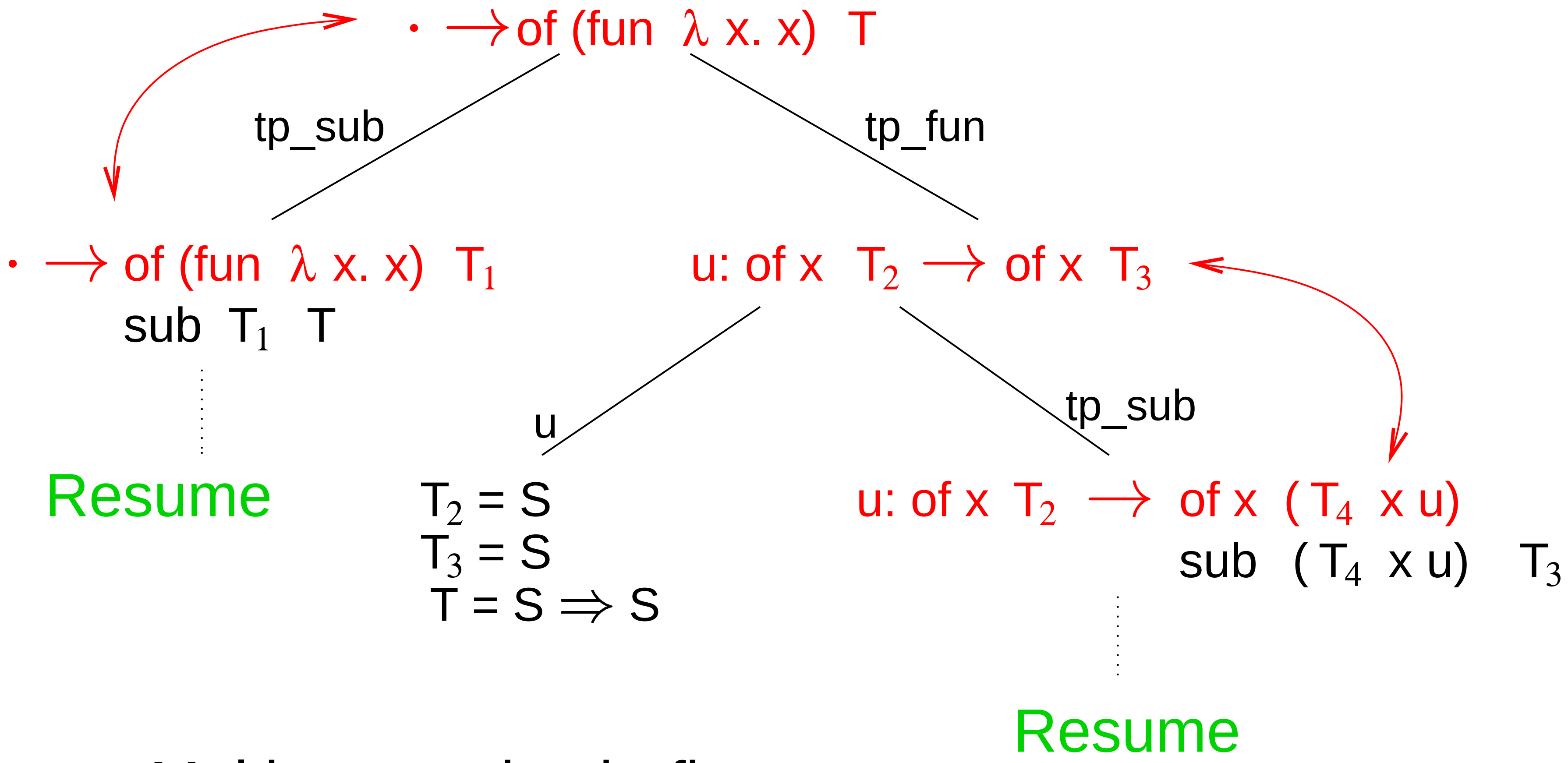


Loop detection

How can we detect loops? Subordination

How can we still produce all answers?

Proof tree (cont.)



Multi−stage depth−first strategy
adapted from [Tamaki, Sato89]

Memoization based proof search

- Proof search using a memo-table
- Store intermediate goals and re-use results
- May need to use subordination!
- Eliminate redundant computation
- Eliminate infinite paths
- More specifications are executable!

Memo-table

- Table entry: ($\Gamma \rightarrow a$, $\mathcal{A}$)
  - Γ : context of assumptions (i.e. u:of x T_2)
  - a : atomic goal (i.e. of (fun λx. x) T, of x T_3)
  - $\mathcal{A}$: list of answer substitutions for all existential variables in Γ and a

Memo-table

- Table entry: ($\Gamma \to a$, $\mathcal{A}$)
  - Γ : context of assumptions (i.e. u:of x T_2)
  - a : atomic goal (i.e. of (fun λx. x) T, of x T_3)
  - $\mathcal{A}$: list of answer substitutions for all existential variables in Γ and a

Goal			Answer
$\cdot$	$\to$	of (fun λx.x) T	T = S $\Rightarrow$ S
u:of x T_2	$\to$	of x T_3	T_2 = S, T_3 = S

Properties

- Selective memoization
- Finds all answers to a query
- Terminates for programs over a finite domain

Theoretical foundation

Conservative extension of LF [Harper *et. al.* 93] with meta-variables

- Foundation for proof search and for other optimization (e.g. higher-order unification, higher-order term indexing)
- Type-checking remains decidable.
- Canonical forms exist.
- Proofs follow [Harper,Pfenning03]

Tabled proof search

Uniform proofs as a foundation for logic programming [Miller *et.al* 91]

Soundness Any uniform proof *with answer substitution* has a uniform proof.

Completeness Any uniform proof has a uniform proof *with answer substitution*.

Soundness of tabled higher-order logic programming : Any *tabled* uniform proof with an answer substitution has a uniform proof with the same answer substitution.

Related work

- Related Work: XSB system [Warren et al. 99]
  Very successful for first-order logic programming

- Applicable to other higher-order systems:
  - λProlog[Nadathur,Miller88]
  - Linear logic programming [Hodas et al. 94][Cervesato96]

Outline

- Logical frameworks and certified code
- Tabled higher-order logic programming
  - Basic idea and challenges
  - Experiments and Evaluation
  - Improving efficiency
- Conclusion and future work

Outline

- Logical frameworks and certified code
- Tabled higher-order logic programming
  - Basic idea and challenges
  - Experiments and Evaluation
  - Improving efficiency
- Conclusion and future work

Experiments

- Parsing of formulas (adapted from [Warren99])
  - Left and right recursion
  - Not executable with depth-first search
  - Memoization vs iterative deepening
- Refinement type checking [Davies, Pfenning00]
  - Decidable
  - Memoization vs depth-first search

Parser for formulas

#tok	memo	iterative deepening
20	0.13 sec	0.98 sec
58	2.61 sec	∞
117	10.44 sec	∞
235	75.57 sec	∞

∞ = process does not terminate

Intel Pentium 1.6GHz, RAM 256MB,

SML New Jersey 110, Twelf 1.4

Refinement type-checking

	example	memo	depth-first
First answer	sub		0.15 sec
	mult		0.15 sec
	square		0.16 sec
Not provable	mult		13.50 sec
	plus		∞
	square		∞
All answers	sub		5.59 sec
	mult		∞
	square		∞

Refinement type-checking

	example	memo	depth-first
First answer	sub	3.19 sec	0.15 sec
	mult	7.78 sec	0.15 sec
	square	9.02 sec	0.16 sec
Not provable	mult	2.38 sec	13.50 sec
	plus	6.48 sec	∞
	square	9.29 sec	∞
All answers	sub	6.88 sec	5.59 sec
	mult	9.06 sec	∞
	square	10.30 sec	∞

Evaluation

- Benefits:
  - Superior to iterative deepening
  - Meaningful failure: decision procedure
  - Consistent performance
  - Quick failure
  - Small proof size
- Drawbacks:
  - Overhead of storing and retrieving information
  - Multi-stage strategy delays the reuse of answers

Outline

- Logical frameworks and certified code
- Tabled higher-order logic programming
  - Basic idea and challenges
  - Experiments and Evaluation
  - Improving efficiency
- Conclusion and future work

Outline

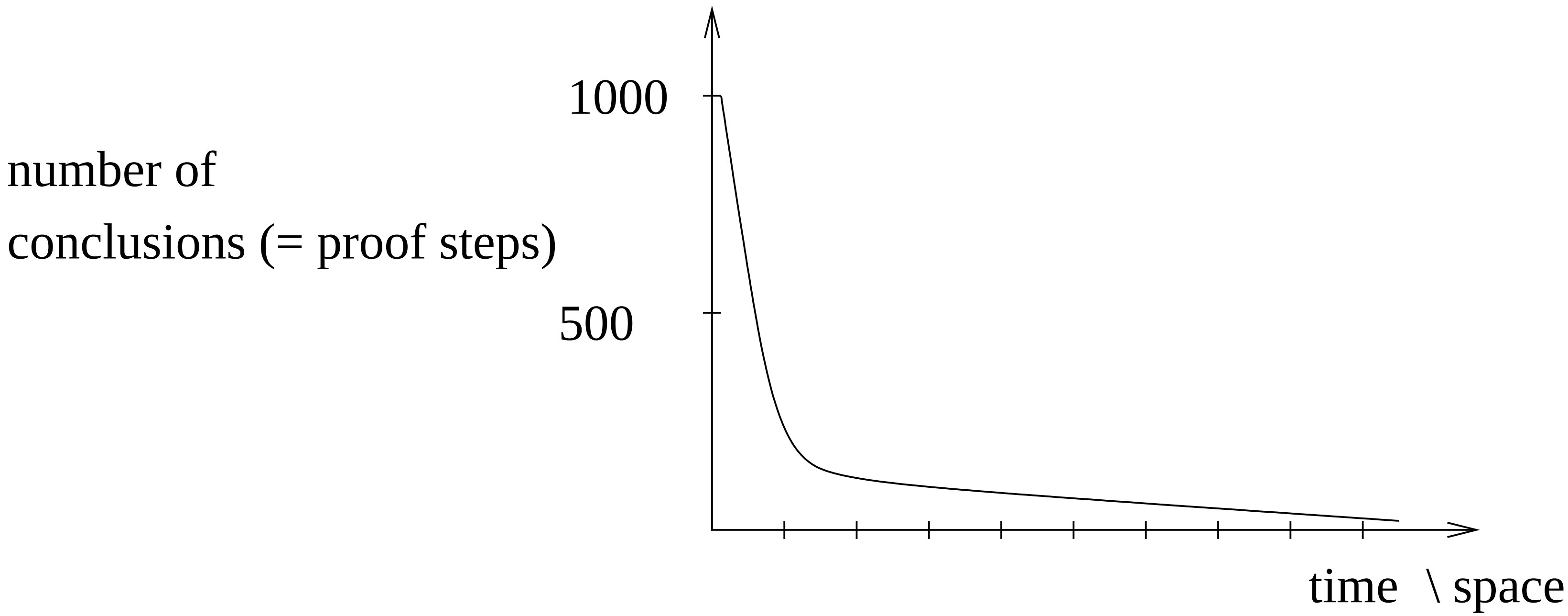
- Logical frameworks and certified code
- Tabled higher-order logic programming
  - Basic idea and challenges
  - Experiments and Evaluation
  - Improving efficiency
- Conclusion and future work

Efficiently accessing the memo-table

“...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]

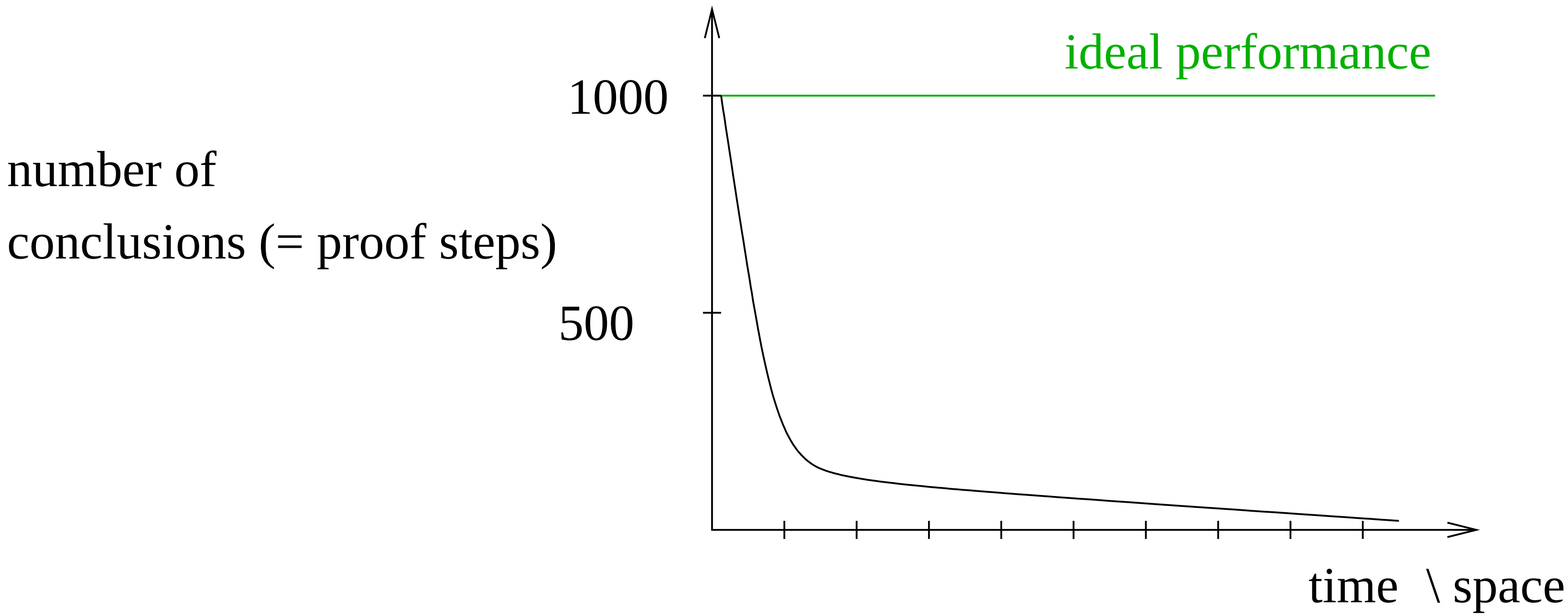
Efficiently accessing the memo-table

"...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]



Efficiently accessing the memo-table

"...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

(1) pred (h (h b)) (g b) (f λx. E x)

(2) pred (h (h a)) (g b) (f λx. E x)

(3) pred (h (g a)) (g b) a

Query:

pred (h (h b)) (g b) a

How can we efficiently store and retrieve data?

Indexing

Set of terms

(1) pred (h (h b)) (g b) (f λx. E x)

(2) pred (h (h a)) (g b) (f λx. E x)

(3) pred (h (g a)) (g b) a

Query:

pred (h (h b)) (g b) a

How can we efficiently store and retrieve data?

- Share term structure
- Share common operations

Common sub-expression

Set of terms

(1) pred (h (h b)) (g b) (f λx. E x)

(2) pred (h (h a)) (g b) (f λx. E x)

(3) pred (h (g a)) (g b) a

Query:

pred (h (h b)) (g b) a

- Factor out common sub-expressions!

pred (h (h a)) (g b) (f λ x. E x)

pred (h (g a)) (g b) a

pred (h *1) (g b) *2

Common sub-expression

Set of terms

(1) pred (h (h b)) (g b) (f λx. E x)

(2) pred (h (h a)) (g b) (f λx. E x)

(3) pred (h (g a)) (g b) a

Query:

pred (h (h b)) (g b) a

- Factor out common sub-expressions!

  pred (h (h a)) (g b) (f λ x. E x)

  pred (h (g a)) (g b) a

  pred (h *1) (g b) *2

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

MSG of higher-order patterns

Set of terms

(1) pred (h (h b)) (g b) (f λx. E x)

(2) pred (h (h a)) (g b) (f λx. E x)

(3) pred (h (g a)) (g b) a

Query:

pred (h (h b)) (g b) a

- Most specific generalization exists for higher-order patterns.
- Not all terms fall within this class.
- Is this efficient?

Our approach

Set of terms

(1) pred (h (h b)) (g b) (f λx. E x)

(2) pred (h (h a)) (g b) (f λx. E x)

(3) pred (h (g a)) (g b) a

Query:

pred (h (h b)) (g b) a

- Further restrict higher-order patterns!
  (Linear higher-order patterns)
  - Every meta-variable occurs only once.
  - Every meta-variable is fully applied.
- Translate terms into linear higher-order patterns and residual equations (variable definitions)

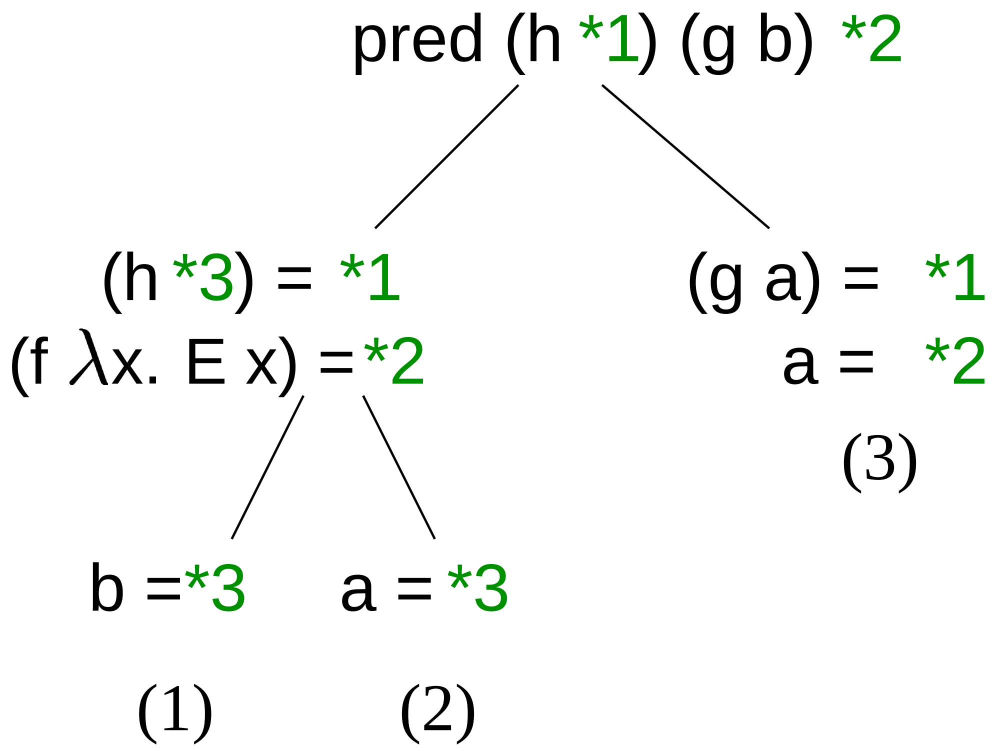
Higher-order substitution trees

Set of terms

(1) pred (h (h b)) (g b) (f λx. E x)

(2) pred (h (h a)) (g b) (f λx. E x)

(3) pred (h (g a)) (g b) a



Compose substitutions!

Parser for formulas

	iterative	memoization		
#tok	deepening	noindex	index	speed-up
20	0.98 sec	0.13 sec	0.07 sec	85%
58	∞	2.61 sec	1.25 sec	108%
117	∞	10.44 sec	5.12 sec	103%
235	∞	75.57 sec	26.08 sec	190%

∞ = process does not terminate

Intel Pentium 1.6GHz, RAM 256MB,

SML New Jersey 110, Twelf 1.4.

Refinement type-checking

	example	noindex	index	speed-up	orig
First	sub	3.19 sec	0.46 sec	593%	
answer	mult	7.78 sec	0.89 sec	774%	
	square	9.02 sec	0.98 sec	820%	
Not	mult	2.38 sec	0.38 sec	526%	
provable	plus	6.48 sec	0.85 sec	662%	
	square	9.29 sec	1.09 sec	752%	
All	sub	6.88 sec	0.71 sec	869%	
answers	mult	9.06 sec	0.98 sec	824%	
	square	10.30 sec	1.08 sec	854%	

Refinement type-checking

	example	noindex	index	speed-up	orig
First	sub	3.19 sec	0.46 sec	593%	0.15 sec
answer	mult	7.78 sec	0.89 sec	774%	0.15 sec
	square	9.02 sec	0.98 sec	820%	0.16 sec
Not	mult	2.38 sec	0.38 sec	526%	13.50 sec
provable	plus	6.48 sec	0.85 sec	662%	∞
	square	9.29 sec	1.09 sec	752%	∞
All	sub	6.88 sec	0.71 sec	869%	5.59 sec
answers	mult	9.06 sec	0.98 sec	824%	∞
	square	10.30 sec	1.08 sec	854%	∞

Contribution and related work

- Contribution:
  - Higher-order term indexing (key: linearization, η-longform)
  - Indexing substantially improves performance
    between 85% and 820%

Contribution and related work

- Contribution:
  - Higher-order term indexing (key: linearization, η-longform)
  - Indexing substantially improves performance between 85% and 820%
- Related Work:
  - Substitution trees for first-order terms [Graf95]
  - (Higher-order) automata-driven indexing [Necula,Rahul01] imperfect filter, calls full higher-order unification to check candidates

Outline

- Logical frameworks and certified code
- Tabled higher-order logic programming
  - Basic idea and challenges
  - Experiments and Evaluation
  - Improving efficiency
- Conclusion and future work

Summary

This talk

- Tabled higher-order logic programming
- Higher-order indexing

In the thesis

- More theory
- Optimizing higher-order unification
- Meta-theorem proving based on tabled higher-order logic programming

Conclusion

- This opens many new opportunities
  - to experiment and develop large-scale systems.
    for example: proof-carrying code
  - 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
  - Improving tabling (e.g. subsumption, different scheduling strategies)
  - Eliminating redundancy in the representation of clauses, goals and proofs: approximate typing [Necula,Lee98]
  - Mode, determinism, termination analysis [Schrijvers et al. 02]
  - Ordered resolution [Bachmair, Ganzinger 01]
  - ...

Theory

- Foundation for meta-variables
  - Abstract over meta-variables ($\Pi^{\Box} u{::}\Psi{\vdash}A.$)
  - First-class variable definitions ($\Pi^{\Box} u = M{::}\Psi{\vdash}A$)
  - Representing and type-checking dag-style objects
- Meta-theorem proving
  - Automating complete induction
  - Further work on redundancy elimination

Applications

Proof-carrying code

- How can we transmit small proofs?[Necula,Rahul 01], (collaboration with Crary and Sarkar)
- How can we check them efficiently? [Stump, Dill 02]
- How can we automate some of the meta-proofs? [Crary,Sarkar03]

Applications

Proof-carrying code

- How can we transmit small proofs?[Necula,Rahul 01], (collaboration with Crary and Sarkar)
- How can we check them efficiently? [Stump, Dill 02]
- How can we automate some of the meta-proofs? [Crary,Sarkar03]

Proof-carrying authorization [Bauer et al. 02]

Bob proves that he is authorized to access Alice's web-page.

- How can we efficiently generate proofs?
- How can we cache and re-use proof attempts?

Finally ...

The End.

Finally ...

The End.

if you want to find out more:

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