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 are plentiful in computer science.

- Axioms and inference rules
- Examples: operational semantics, type system, logic, etc.
Deductive systems and logical frameworks

Deductive systems are plentiful in 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 \( \tau ::= \) zero | pos | nat | bit | \( \tau_1 \Rightarrow \tau_2 \) | \ldots
```

Example: \( 6 = \epsilon_{110} \) and \( \epsilon_{110} \in \text{nat} \)
Declarative description of subtyping

types \( \tau ::= \) zero | pos | nat | bit | \( \tau_1 \Rightarrow \tau_2 \) | \ldots

Example: \( 6 = \epsilon_{110} \) and \( \epsilon_{110} \in \text{nat} \)

\[
\begin{align*}
\text{zero} & \prec \text{nat} \\
\text{pos} & \prec \text{nat} \\
\text{nat} & \prec \text{bit}
\end{align*}
\]
Declarative description of subtyping

types $\tau ::= \text{zero} | \text{pos} | \text{nat} | \text{bit} | \tau_1 \Rightarrow \tau_2 | \ldots$

Example: $6 = \epsilon_{110}$ and $\epsilon_{110} \in \text{nat}$

$\text{zero} \preceq \text{nat}$  
$\text{pos} \preceq \text{nat}$  
$\text{nat} \preceq \text{bit}$  

$\tau \preceq \tau$  

$\tau_1 \preceq \tau_2$  

Tabled higher-order logic programming – p.4/47
Typing rules for Mini-ML

expressions  \( e ::= \varepsilon \mid e\ 0 \mid e\ 1 \mid \text{fun } x.e \mid \text{app } e_1\ e_2 \)

\[
\Gamma \vdash e : \tau' \quad \tau' \leq \tau \\
\frac{}{\Gamma \vdash e : \tau} \quad \text{tp-sub}
\]

\[
\Gamma, x:\tau_1 \vdash \tau_2: \quad \frac{}{\Gamma \vdash \text{fun } x.e : \tau_1 \Rightarrow \tau_2} \quad \text{tp-fun}
\]
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.  \texttt{?-} 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. ?- sub zero bit.

nb: sub nat bit.

refl: sub T T.

tr: sub T S

<- sub T R

<- sub R S.

Proof: (tr nb zn)

yes
Implementation of typing rules

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

tp_fun: of (fun λx.E x) (T1 => T2)
     <- (∏ x: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:
  - $\lambda$-abstraction
  - dependent types
- Dynamic program clauses
- Explicit proof objects
Higher-order logic programming

- Higher-order data-types:
  - $\lambda$-abstraction
  - dependent types
- Dynamic program clauses
- Explicit proof objects

Different approaches: $\lambda$Prolog, Isabelle, Twelf
Application: certified code

- Foundational proof-carrying code: [Appel, Felty 00]
- Temporal-logic proof carrying code: [Bernard, Lee 02]
- Foundational typed assembly language: [Crary 03]
- Proof-carrying authentication: [Felten, Appel 99]
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!
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
• Logical frameworks and certified code
• Tabled higher-order logic programming
  - Basic idea and challenges
  - Experiments and Evaluation
  - Improving efficiency
• Conclusion and future work
“...it is very common for the proofs to have repeated sub-proofs that should be hoisted out and proved only once ...” [Necula,Lee97]
“...it is very common for the proofs to have repeated sub-proofs that should be hoisted out and proved only once ...” [Necula, Lee97]
“...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 λ x.E x) (T1 => T2)
       <- (∏ x:exp.of x T1 -> of (E x) T2).
       "forall x:exp, assume of x T1
        and show of (E x) T2"
\[
\begin{align*}
\text{Proof tree} & \quad \cdot \rightarrow \text{of (fun } \lambda x. x) \ T \\
& \quad \quad \text{tp_sub} \quad \text{tp_fun} \\
& \quad \cdot \rightarrow \text{of (fun } \lambda x. x) \ T_1 \quad u : \text{of } x \ T_2 \rightarrow \text{of } x \ T_3 \\
& \quad \quad \text{sub } T_1 \ T \\
& \quad \quad u \quad \text{tp_sub} \\
& \quad \quad T_2 = S \quad \quad u : \text{of } x \ T_2 \rightarrow \text{of } x \ (T_4 \ x \ u) \\
& \quad \quad T_3 = S \quad \quad \text{sub } (T_4 \ x \ u) \ T_3 \\
& \quad \quad T = S \Rightarrow S
\end{align*}
\]
Proof tree

Loop detection
Proof tree

How can we detect loops?

Loop detection
Loops modulo strengthening

- Dependencies among terms
  \[ u:of \ x \ T_2 \rightarrow of \ x \ (T_4 \ x \ u) \]
Loops modulo strengthening

- Dependencies among terms
  \[ u:of \times T_2 \rightarrow of \times (T_4 \times u) \]
  
  strengthen \[ u:of \times T_2 \rightarrow of \times T_4 \]
Loops modulo strengthening

- Dependencies among terms
  \[ u : \text{of} \times T_2 \rightarrow \text{of} \times (T_4 \times u) \]
  strengthen \[ u : \text{of} \times T_2 \rightarrow \text{of} \times T_4 \]

- Dependencies among propositions
  \[ u : \text{of} \times T_2 \rightarrow \text{sub} (T_4 \times u) \times T_3 \]
Loops modulo strengthening

- Dependencies among terms
  \[ u:of \times T_2 \rightarrow of \times (T_4 \times u) \]
  strengthen \[ u:of \times T_2 \rightarrow of \times T_4 \]

- Dependencies among propositions
  \[ u:of \times T_2 \rightarrow sub (T_4 \times u) T_3 \]
  strengthen: \[ \cdot \rightarrow sub T_4 T_3 \]
Loops modulo strengthening

- Dependencies among terms
  \[ u:\text{of} x \ T_2 \rightarrow \text{of} x \ (T_4 \times u) \]
  strengthen \[ u:\text{of} x \ T_2 \rightarrow \text{of} x \ T_4 \]

- Dependencies among propositions
  \[ u:\text{of} x \ T_2 \rightarrow \text{sub} (T_4 \times u) \ T_3 \]
  strengthen: \[ \rightarrow \text{sub} T_4 \ T_3 \]

- Subordination analysis [Virga99]
Loop detection
How can we detect loops?
Proof tree (cont.)

How can we detect loops?

Subordination

Loop detection

u: of x $\rightarrow$ of x $T_2$ $\rightarrow$ of x $(T_4 \times u)$

sub $(T_4 \times u)$ $T_3$

$T_2 = S$
$T_3 = S$
$T = S \Rightarrow S$

$\cdot \rightarrow \text{of } (\text{fun } \lambda x. x) \ T$

$\cdot \rightarrow \text{of } (\text{fun } \lambda x. x) \ T_1$

sub $T_1$ $T$

$\text{tp}_\text{sub}$

$\text{tp}_\text{fun}$

$\text{tp}_\text{sub}$
Loop detection
How can we detect loops? **Subordination**

How can we still produce all answers?
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: \((\Gamma \rightarrow a, \mathcal{A})\)
  - \(\Gamma\): context of assumptions (i.e. \(u:of x T_2\))
  - \(a\): atomic goal (i.e. of (fun \(\lambda x. x\) \(T\), of \(x T_3\))
  - \(\mathcal{A}\): list of answer substitutions for all existential variables in \(\Gamma\) and \(a\)
**Memo-table**

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

<table>
<thead>
<tr>
<th>Goal</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\cdot \rightarrow \text{of } (\text{fun } \lambda x. x) T$</td>
<td>$T = S \Rightarrow S$</td>
</tr>
<tr>
<td>$u:\text{of } x T_2 \rightarrow \text{of } x T_3$</td>
<td>$T_2 = S, T_3 = S$</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>#tok</th>
<th>memo</th>
<th>iterative deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.13 sec</td>
<td>0.98 sec</td>
</tr>
<tr>
<td>58</td>
<td>2.61 sec</td>
<td>∞</td>
</tr>
<tr>
<td>117</td>
<td>10.44 sec</td>
<td>∞</td>
</tr>
<tr>
<td>235</td>
<td>75.57 sec</td>
<td>∞</td>
</tr>
</tbody>
</table>

∞ = process does not terminate

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

<table>
<thead>
<tr>
<th></th>
<th>example</th>
<th>memo</th>
<th>depth-first</th>
</tr>
</thead>
<tbody>
<tr>
<td>First answer</td>
<td>sub</td>
<td></td>
<td>0.15 sec</td>
</tr>
<tr>
<td></td>
<td>mult</td>
<td></td>
<td>0.15 sec</td>
</tr>
<tr>
<td></td>
<td>square</td>
<td></td>
<td>0.16 sec</td>
</tr>
<tr>
<td>Not provable</td>
<td>mult</td>
<td></td>
<td>13.50 sec</td>
</tr>
<tr>
<td></td>
<td>plus</td>
<td></td>
<td>∞</td>
</tr>
<tr>
<td></td>
<td>square</td>
<td></td>
<td>∞</td>
</tr>
<tr>
<td>All answers</td>
<td>sub</td>
<td></td>
<td>5.59 sec</td>
</tr>
<tr>
<td></td>
<td>mult</td>
<td></td>
<td>∞</td>
</tr>
<tr>
<td></td>
<td>square</td>
<td></td>
<td>∞</td>
</tr>
</tbody>
</table>
## Refinement type-checking

<table>
<thead>
<tr>
<th>Example</th>
<th>Memo</th>
<th>Depth-first</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>First answer</strong></td>
<td>sub</td>
<td>3.19 sec</td>
</tr>
<tr>
<td></td>
<td>mult</td>
<td>7.78 sec</td>
</tr>
<tr>
<td></td>
<td>square</td>
<td>9.02 sec</td>
</tr>
<tr>
<td><strong>Not provable</strong></td>
<td>mult</td>
<td>2.38 sec</td>
</tr>
<tr>
<td></td>
<td>plus</td>
<td>6.48 sec</td>
</tr>
<tr>
<td></td>
<td>square</td>
<td>9.29 sec</td>
</tr>
<tr>
<td><strong>All answers</strong></td>
<td>sub</td>
<td>6.88 sec</td>
</tr>
<tr>
<td></td>
<td>mult</td>
<td>9.06 sec</td>
</tr>
<tr>
<td></td>
<td>square</td>
<td>10.30 sec</td>
</tr>
</tbody>
</table>
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
“...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) \text{pred} (h (h \text{b})) (g \text{b}) (f \lambda x. E x)
(2) \text{pred} (h (h \text{a})) (g \text{b}) (f \lambda x. E x)
(3) \text{pred} (h (g \text{a})) (g \text{b}) \text{a}

Query:

pred (h (h \text{b})) (g \text{b}) \text{a}

How can we efficiently store and retrieve data?
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) $\text{pred} (h (h \ b)) \ (g \ b) \ (f \ \lambda x. \ E \ x)$
(2) $\text{pred} (h (h \ a)) \ (g \ b) \ (f \ \lambda x. \ E \ x)$
(3) $\text{pred} (h (g \ a)) \ (g \ b) \ a$

Query:
$\text{pred} (h (h \ b)) \ (g \ b) \ a$

- Factor out common sub-expressions!
  $\text{pred} (h (h \ a)) \ (g \ b) \ (f \ \lambda x. \ E \ x)$
  $\text{pred} (h (g \ a)) \ (g \ b) \ a$

  $\text{pred} (h ^1) \ (g \ b) ^2$
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 λx. E x)
pred (h (g a)) (g b) a

- 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 \(\lambda x. E x\))
(2) pred (h (h a)) (g b) (f \(\lambda x. E x\))
(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) \( \text{pred} (h (h b)) (g b) (f \lambda x. E x) \)
(2) \( \text{pred} (h (h a)) (g b) (f \lambda x. E x) \)
(3) \( \text{pred} (h (g a)) (g b) a \)

Compose substitutions!

(1) \( (h *3) = *1 \)
(2) \( (f \lambda x. E x) = *2 \)
(3) \( (g a) = *1 \)
(4) \( a = *2 \)
(5) \( b = *3 \)

\( \text{pred} (h *1) (g b) *2 \)
### Parser for formulas

<table>
<thead>
<tr>
<th>#tok</th>
<th>iterative deepen</th>
<th>memoization noindex</th>
<th>memoization index</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.98 sec</td>
<td>0.13 sec</td>
<td>0.07 sec</td>
<td>85%</td>
</tr>
<tr>
<td>58</td>
<td>∞</td>
<td>2.61 sec</td>
<td>1.25 sec</td>
<td>108%</td>
</tr>
<tr>
<td>117</td>
<td>∞</td>
<td>10.44 sec</td>
<td>5.12 sec</td>
<td>103%</td>
</tr>
<tr>
<td>235</td>
<td>∞</td>
<td>75.57 sec</td>
<td>26.08 sec</td>
<td>190%</td>
</tr>
</tbody>
</table>

∞ = process does not terminate

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>speed-up</th>
<th>orig</th>
</tr>
</thead>
<tbody>
<tr>
<td>First sub</td>
<td>3.19 sec</td>
<td>0.46 sec</td>
<td>593%</td>
<td></td>
</tr>
<tr>
<td>answer</td>
<td>7.78 sec</td>
<td>0.89 sec</td>
<td>774%</td>
<td></td>
</tr>
<tr>
<td>square</td>
<td>9.02 sec</td>
<td>0.98 sec</td>
<td>820%</td>
<td></td>
</tr>
<tr>
<td>Not mult</td>
<td>2.38 sec</td>
<td>0.38 sec</td>
<td>526%</td>
<td></td>
</tr>
<tr>
<td>provable plus</td>
<td>6.48 sec</td>
<td>0.85 sec</td>
<td>662%</td>
<td></td>
</tr>
<tr>
<td>square</td>
<td>9.29 sec</td>
<td>1.09 sec</td>
<td>752%</td>
<td></td>
</tr>
<tr>
<td>All sub</td>
<td>6.88 sec</td>
<td>0.71 sec</td>
<td>869%</td>
<td></td>
</tr>
<tr>
<td>answers mult</td>
<td>9.06 sec</td>
<td>0.98 sec</td>
<td>824%</td>
<td></td>
</tr>
<tr>
<td>square</td>
<td>10.30 sec</td>
<td>1.08 sec</td>
<td>854%</td>
<td></td>
</tr>
</tbody>
</table>
## Refinement type-checking

<table>
<thead>
<tr>
<th>example</th>
<th>noindex</th>
<th>index</th>
<th>speed-up</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>593%</td>
</tr>
<tr>
<td>answer</td>
<td>mult</td>
<td>7.78 sec</td>
<td>0.89 sec</td>
<td>774%</td>
</tr>
<tr>
<td>square</td>
<td></td>
<td>9.02 sec</td>
<td>0.98 sec</td>
<td>820%</td>
</tr>
<tr>
<td>Not</td>
<td>mult</td>
<td>2.38 sec</td>
<td>0.38 sec</td>
<td>526%</td>
</tr>
<tr>
<td>provable</td>
<td>plus</td>
<td>6.48 sec</td>
<td>0.85 sec</td>
<td>662%</td>
</tr>
<tr>
<td>square</td>
<td></td>
<td>9.29 sec</td>
<td>1.09 sec</td>
<td>752%</td>
</tr>
<tr>
<td>All</td>
<td>sub</td>
<td>6.88 sec</td>
<td>0.71 sec</td>
<td>869%</td>
</tr>
<tr>
<td>answers</td>
<td>mult</td>
<td>9.06 sec</td>
<td>0.98 sec</td>
<td>824%</td>
</tr>
<tr>
<td>square</td>
<td></td>
<td>10.30 sec</td>
<td>1.08 sec</td>
<td>854%</td>
</tr>
</tbody>
</table>
Contribution and related work

- Contribution:
  - Higher-order term indexing (key: linearization, $\eta$-longform)
  - Indexing substantially improves performance between 85% and 820%
Contribution and related work

• Contribution:
  – Higher-order term indexing (key: linearization, $\eta$-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
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, Sarkar 03]
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, Sarkar 03]

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