Memoization-based proof search in LF: an experimental evaluation of a prototype

Brigitte Pientka

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA, 15217, USA
email: bp@cs.cmu.edu
Outline

• LF as a logic programming framework
• Example: Type-system with subtyping
• Basics of tabled higher-order logic programming
• Experimental evaluation:
  1. Refinement type-checking: Depth-first vs tabled search
  2. Parsing into higher-order abstract syntax: Iterative deepening vs tabled search
• Related Work
• Conclusion and future work
Logical framework LF [Harper,Honsell,Plotkin93] dependently typed λ-calculus
Logical framework LF [Harper, Honsell, Plotkin93] dependently typed $\lambda$-calculus

Framework for specifying and implementing
  • logical systems
  • proofs about them
LF as a logic programming framework

Logical framework LF [Harper, Honsell, Plotkin93] dependently typed λ-calculus

Framework for specifying and implementing
- logical systems (safety logics, type system . . .)
- proofs about them
Logical framework LF [Harper, Honsell, Plotkin 1993] dependently typed \(\lambda\)-calculus

Framework for specifying and implementing

- logical systems (safety logics, type system ...)
- proofs about them (correctness, soundness ...)

Memoization-based proof search in LF: an experimental evaluation of a prototype – p.3/34
Logical framework LF [Harper,Honsell,Plotkin93] dependently typed $\lambda$-calculus

Framework for specifying and implementing
- logical systems (safety logics, type system . . . )
- proofs about them (correctness, soundness . . . )

Proof search via higher-order logic programming [Pfenning91]
- Terms: (dependently) typed $\lambda$-calculus
- Clauses: implication, universal quantification
Proof search over declarative systems

Proof search problems:
  • Infinite computation leads to non-termination. \(\Rightarrow\) many specifications are not executable
  • Redundant computation hampers performance.

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

Proof search problems:

- Infinite computation leads to non-termination.
  \[ \Rightarrow \text{many specifications are not executable} \]
- Redundant computation hampers performance.

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

Solution: Memoization and re-use of sub-proofs
Memoization-based proof search

First-order tabelling [Tamaki, Sato 86]

- Memoize atomic subgoals and re-use results
- Finds all possible answers to a query
- Terminates for programs in a finite domain
- Combine tabled and non-tabled execution
- Very successful: XSB system [Warren et al.]
Memoization-based proof search

First-order tabelling [Tamaki, Sato86]
- Memoize atomic subgoals and re-use results
- Finds all possible answers to a query
- Terminates for programs in a finite domain
- Combine tabled and non-tabled execution
- Very successful: XSB system [Warren et.al.]

Higher-order tabelling (see also [Pientka, ICLP’02])
- Proof-theoretic characterization
- This talk: Experiments with higher-order tabling
Outline

• LF as a logic programming framework
• Example: Type-system with subtyping
• Basics of tabled higher-order logic programming
• Experimental evaluation:
  1. Refinement type-checking:
     Depth-first vs tabled search
  2. Parsing into higher-order abstract syntax:
     Iterative deepening vs tabled search
• Related Work
• Conclusion and future work
Declarative description of subtyping

\[
types \quad \tau :: = \quad \text{zero} \mid \text{pos} \mid \text{nat} \mid \text{bit} \mid \tau_1 \Rightarrow \tau_2 \mid \ldots
\]

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

Types \( \tau :: = \) zero | pos | nat | bit | \( \tau_1 \Rightarrow \tau_2 \) | ... 

Example: 6 = 110 and 110 ∈ nat

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

\[ \tau ::= \text{zero} \mid \text{pos} \mid \text{nat} \mid \text{bit} \mid \tau_1 \Rightarrow \tau_2 \mid \ldots \]

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

\[
\begin{align*}
\text{zero} \preceq \text{nat} & \quad \text{pos} \preceq \text{nat} & \quad \text{nat} \preceq \text{bit} \\
T \preceq T & \quad T \preceq R & \quad R \preceq S \\
\end{align*}
\]

\[
T \preceq S \quad \text{tr}
\]
Typing rules for Mini-ML

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

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

\[
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{lam } x.e : \tau_1 \Rightarrow \tau_2} \quad \text{tp-lam}\ ^{x}
\]
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.
nb: sub nat bit.
refl: sub T T.
tr: sub T S
<- sub T R
<- sub R S.

Not executable!
Implementation of typing rules

\[ \text{tp}_{\text{sub}} : \text{of } E \ T \]
\[ \quad \leftarrow \text{of } E \ T' \]
\[ \quad \leftarrow \text{sub } T' \ T. \]

\[ \text{tp}_{\text{lam}} : \text{of } (\text{lam } \lambda x. E \ x) \ (T1 \Rightarrow T2) \]
\[ \quad \leftarrow (\Pi x:exp. \text{of } x \ T1 \Rightarrow \text{of } (E \ x) \ T2). \]
\[ \quad \text{“forall } x:exp, \text{ assume of } x \ T1 \]
\[ \quad \text{ and show of } (E \ x) \ T2” \]
Implementation of typing rules

\[
\text{tp\_sub}: \quad \text{of } E \ T \\
\quad \Rightarrow \text{of } E \ T' \\
\quad \Rightarrow \text{sub } T' \ T.
\]

\[
\text{tp\_lam}: \quad \text{of } (\text{lam } \lambda x.E \ x) \ (T1 \Rightarrow T2) \\
\quad \Rightarrow (\Pi x:exp.\text{of } x \ T1 \Rightarrow \text{of } (E \ x) \ T2). \\
\quad \text{“forall } x:exp, \text{assume of } x \ T1 \text{ and show of } (E \ x) \ T2”
\]

Redundancy: \text{tp\_sub} is always applicable!
Outline

• LF as a logic programming framework
• Example: Type-system with subtyping
• Basics of tabled higher-order logic programming
• Experimental evaluation:
  1. Refinement type-checking:
     Depth-first vs tabled search
  2. Parsing into higher-order abstract syntax:
     Iterative deepening vs tabled search
• Related Work
• Conclusion and future work
Tabled higher-order logic programming

- Eliminate redundant and infinite paths from proof search using a memo-table

- Table entry: \((\Gamma \rightarrow a, A)\)
  - \(\Gamma\): context of assumptions (i.e. \(x:exp, u:of \ x \ T1\))
  - \(a\): atomic goal (i.e. \(of (lam \ x. \ x) \ T\))
  - \(A\): list of answer substitutions for all free variables in \(\Gamma\) and \(a\)

- Depth-first multi-stage strategy adopted from [Tamaki, Sato89]
How higher-order tabling works...

Stage 1

· \[ \rightarrow \text{of (lam } \lambda x.x) T \]

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How higher-order tabling works...

Stage 1

\[ \cdot \to \text{of} (\text{lam } \lambda x.x) \ T \]

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \cdot \to \text{of} (\text{lam } \lambda x.x) \ T ]</td>
<td></td>
</tr>
</tbody>
</table>
How higher-order tabling works...

Stage 1

\[ \cdot \rightarrow \text{of (lam } \lambda \text{x.x) } T \]

\[ \text{tp_sub} \quad \rightarrow \quad \cdot \rightarrow \text{of (lam } \lambda \text{x.x) R,} \]
\[ \text{sub R T} \]

Entry | Answers
--- | ---
\[ \cdot \rightarrow \text{of (lam } \lambda \text{x.x) } T \]
How higher-order tabling works...

Stage 1

\[ \cdot \rightarrow \text{of } (\text{lam } \lambda x.x) \ T \]

\[ \text{tp_sub} \]

\[ \cdot \rightarrow \text{of } (\text{lam } \lambda x.x) \ R, \]

\[ \text{sub } R \ T \]

Suspend

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \cdot \rightarrow \text{of } (\text{lam } \lambda x.x) \ T ]</td>
<td></td>
</tr>
</tbody>
</table>
How higher-order tabling works...

Stage 1

\[ \rightarrow \text{ of } (\text{lam } \lambda x.x) \ T \]

\[ \text{tp}_\text{sub} \]

\[ \rightarrow \text{ of } (\text{lam } \lambda x.x) \ R, \]

\[ \text{sub } R \ T \]

\[ \text{Suspend} \]

\[ \text{tp}_\text{lam} \]

\[ x: \text{exp}, u: \text{of } x \ T_1 \rightarrow \text{ of } x \ T_2 \]

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answers</th>
</tr>
</thead>
</table>
| \[ \rightarrow \text{ of } (\text{lam } \lambda x.x) \ T \] | }
How higher-order tabling works...

Stage 1

\[ \cdot \rightarrow \text{of (lam } \lambda x.x) \ T \]

\[ \text{tp\_sub} \quad \cdot \rightarrow \text{of (lam } \lambda x.x) \ R, \text{Suspend} \]

\[ \text{tp\_lam} \quad \text{sub} \ R \ T \]

\[ x: \text{exp}, u: \text{of } x \ T_1 \rightarrow \text{of } x \ T_2 \]

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \cdot \rightarrow \text{of (lam } \lambda x.x) \ T ]</td>
<td></td>
</tr>
<tr>
<td>[ x: \text{exp}, u: \text{of } x \ T_1 \rightarrow \text{of } x \ T_2 ]</td>
<td></td>
</tr>
</tbody>
</table>
How higher-order tabling works...

Stage 1

\[ \rightarrow \text{of (lam } \lambda x.x) \ T \]

\[ \text{tp_sub} \rightarrow \]

\[ \rightarrow \text{of (lam } \lambda x.x) \ R, \]

\[ \text{sub } R \ T \]

\[ \text{Suspend} \]

\[ \text{tp_lam} \rightarrow \]

\[ x:\text{exp}, \ u:\text{of } x \ T_1 \rightarrow \text{of } x \ T_2 \quad u \]

\[ \text{T}_1 = S, \ T_2 = S, \ T = (S \Rightarrow S) \]

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \rightarrow \text{of (lam } \lambda x.x) \ T ]</td>
<td></td>
</tr>
<tr>
<td>[ x:\text{exp}, \ u:\text{of } x \ T_1 \rightarrow \text{of } x \ T_2 ]</td>
<td></td>
</tr>
</tbody>
</table>
How higher-order tabling works...

Stage 1

\[ \cdot \rightarrow \text{of } (\text{lam } \lambda x.x) \ T \]

\[ \text{tp}_\text{sub} \quad \cdot \rightarrow \text{of } (\text{lam } \lambda x.x) \ R, \]
\[ \text{sub } R \ T \]

\[ \text{Suspend} \]

\[ \text{tp}_\text{lam} \quad x: \text{exp}, u: \text{of } x \ T_1 \rightarrow \text{of } x \ T_2 \]
\[ u \quad T_1 = S, \ T_2 = S, \ T = (S \Rightarrow S) \]

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \cdot \rightarrow \text{of } (\text{lam } \lambda x.x) \ T ]</td>
<td>[ T = (S \Rightarrow S) ]</td>
</tr>
<tr>
<td>[ x: \text{exp}, u: \text{of } x \ T_1 \rightarrow \text{of } x \ T_2 ]</td>
<td>[ T_1 = S, \ T_2 = S ]</td>
</tr>
</tbody>
</table>
How higher-order tabling works...

Stage 1

\[ \cdot \rightarrow \text{of (lam } \lambda x.x) \ T \]

\[ \text{tp_sub} \quad \cdot \rightarrow \text{of (lam } \lambda x.x) R, \]
\[ \quad \text{sub R } T \]

\[ \text{Suspend} \]

\[ \text{tp_lam} \quad x: \text{exp}, u: \text{of x } T_1 \rightarrow \text{of x } T_2 \]

\[ \text{u} \quad T_1 = S, \quad T_2 = S, \quad T = (S \Rightarrow S) \]

\[ \text{tp_sub} \quad x: \text{exp}, u: \text{of x } T_1 \rightarrow \text{of x } R, \]
\[ \quad \text{sub R } T_2 \]

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>\cdot \rightarrow \text{of (lam } \lambda x.x) \ T</td>
<td>T = (S \Rightarrow S)</td>
</tr>
<tr>
<td>x: \text{exp}, u: \text{of x } T_1 \rightarrow \text{of x } T_2</td>
<td>T_1 = S, \quad T_2 = S</td>
</tr>
</tbody>
</table>
How higher-order tabling works...

Stage 1

\[ \cdot \rightarrow \text{of (lam } \lambda x.x) \ T \]

\[
\begin{array}{l}
\text{tp}._\text{sub} \quad \cdot \rightarrow \text{of (lam } \lambda x.x) \ R, \\
\text{sub} \ R \ T \\
\text{tp}._\text{lam} \quad x: \text{exp}, \ u: \text{of} \ x \ T_1 \rightarrow \text{of} \ x \ T_2 \\
\end{array}
\]

Suspend

\[ u \rightarrow T_1 = S, \ T_2 = S, \ T = (S \Rightarrow S) \]

\[
\begin{array}{l}
\text{tp}._\text{sub} \quad x: \text{exp}, \ u: \text{of} \ x \ T_1 \rightarrow \text{of} \ x \ R, \\
\text{sub} \ R \ T_2 \\
\end{array}
\]

Suspend

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \cdot \rightarrow \text{of (lam } \lambda x.x) \ T ]</td>
<td>[ T = (S \Rightarrow S) ]</td>
</tr>
<tr>
<td>[ x: \text{exp}, \ u: \text{of} \ x \ T_1 \rightarrow \text{of} \ x \ T_2 ]</td>
<td>[ T_1 = S, \ T_2 = S ]</td>
</tr>
</tbody>
</table>

Stage 1 finished
How higher-order tabling works...

Stage 1

\[ \cdot \to \text{of (lam } \lambda x. x) \ T \]

\[ \begin{array}{c}
\text{tp\_lam} \\
\text{tp\_sub}
\end{array} \]

\[ \begin{array}{c}
x: \text{exp}, u: \text{of} \ x \ T_1 \to \text{of} \ x \ T_2 \\
\cdot \to \text{of (lam } \lambda x. x) \ R, \\
\text{sub} \ R \ T
\end{array} \]

\[ \begin{array}{c}
\text{Resume} \\
\text{Suspend}
\end{array} \]

\[ T_1 = S, \ T_2 = S, \ T = (S \Rightarrow S) \]

\[ \begin{array}{c}
\text{tp\_sub} \\
x: \text{exp}, u: \text{of} \ x \ T_1 \to \text{of} \ x \ R, \\
\text{sub} \ R \ T_2
\end{array} \]

Entry | Answers
--- | ---
\[ \cdot \to \text{of (lam } \lambda x. x) \ T \] | \[ T = (S \Rightarrow S) \]
\[ x: \text{exp}, u: \text{of} \ x \ T_1 \to \text{of} \ x \ T_2 \] | \[ T_1 = S, \ T_2 = S \]

Stage 1 finished
Stage 1

\[ \cdot \rightarrow \text{of}(\text{lam } \lambda x.x) \text{T} \]

\[ \begin{array}{c}
\text{tp_sub} \\
\text{tp_lam}
\end{array} \]

\[ \cdot \rightarrow \text{of}(\text{lam } \lambda x.x) \text{R}, \]
\[ \text{sub R T} \]

\[ \text{Suspend} \]

\[ \text{Resume} \]

\[ \begin{array}{c}
x:\text{exp}, u:\text{of x T1} \rightarrow \text{of x T2} \\
\text{Resume}
\end{array} \]

\[ T_1 = S, T_2 = S, T = (S \Rightarrow S) \]

\[ \text{Suspend} \]

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \cdot \rightarrow \text{of}(\text{lam } \lambda x.x) \text{T} ]</td>
<td>[ T = (S \Rightarrow S) ]</td>
</tr>
<tr>
<td>[ x:\text{exp}, u:\text{of x T1} \rightarrow \text{of x T2} ]</td>
<td>[ T_1 = S, T_2 = S ]</td>
</tr>
</tbody>
</table>

Stage 1 finished
Higher-order issues

- Dependencies among propositions
  \[ x:exp, u:of\ x\ P \rightarrow\ sub\ P\ R \]
Higher-order issues

• Dependencies among propositions
  \[ x:\text{exp}, u:\text{of} \ x \ P \rightarrow \text{sub} \ P \ R, \]
  strengthen:
  \[ \rightarrow \text{sub} \ P \ R \]
Higher-order issues

• Dependencies among propositions
  \[ x: \text{exp}, u: \text{of} \ x \ P \rightarrow \text{sub} \ P \ R, \]
  strengthen:
  \[ \rightarrow \text{sub} \ P \ R \]

• Dependencies among terms
  \[ x: \text{exp}, u: \text{of} \ x \ T1 \rightarrow \text{of} \ x \ (R \ x \ u), \]
Higher-order issues

• Dependencies among propositions
  \[ x:exp, u:of \ x \ P \rightarrow \ sub \ P \ R, \]
  strengthen:
  \[ \rightarrow \ sub \ P \ R \]

• Dependencies among terms
  \[ x:exp, u:of \ x \ T1 \rightarrow of \ x \ (R \ x \ u), \]
  strengthen \[ x:exp, u:of \ x \ T1 \rightarrow of \ x \ R \]
Higher-order issues

- Dependencies among propositions
  \[ x: \text{exp}, u: \text{of } x \ P \rightarrow \text{sub } P \ R, \]
  strengthen:
  \[\rightarrow \text{sub } P \ R\]

- Dependencies among terms
  \[ x: \text{exp}, u: \text{of } x \ T1 \rightarrow \text{of } x \ (R \ x \ u), \]
  strengthen
  \[ x: \text{exp}, u: \text{of } x \ T1 \rightarrow \text{of } x \ R \]

- Subordination analysis [Virga99]
Outline

• LF as a logic programming framework
• Example: Type-system with subtyping
• Basics of tabled higher-order logic programming
• Experimental evaluation:
  1. Refinement type-checking: Depth-first vs tabled search
  2. Parsing into higher-order abstract syntax: Iterative deepening vs tabled search
• Related Work
• Conclusion and future work
Outline

• LF as a logic programming framework
• Example: Type-system with subtyping
• Basics of tabled higher-order logic programming
• Experimental evaluation:
  1. Refinement type-checking: Depth-first vs tabled search
  2. Parsing into higher-order abstract syntax: Iterative deepening vs tabled search
• Related Work
• Conclusion and future work
Refinement type checking

- Type-inference with subtyping and intersections
- Bi-directional type-checking algorithm [Davies, Pfenning00]
- Distinguish between expressions for which
  1. a type can be synthesized
  2. can be checked against a given type
## Depth-first vs Memoization (all solutions)

<table>
<thead>
<tr>
<th>Program</th>
<th>Depth-First</th>
<th>Memoization</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus’4</td>
<td>483.070 sec</td>
<td>2.330 sec</td>
</tr>
<tr>
<td>plus4</td>
<td>696.730 sec</td>
<td>3.150 sec</td>
</tr>
<tr>
<td>plus4(np)</td>
<td>22.770 sec</td>
<td>1.95 sec</td>
</tr>
<tr>
<td>sub’1a</td>
<td>0.070 sec</td>
<td>0.240 sec</td>
</tr>
<tr>
<td>sub1b</td>
<td>3.88 sec</td>
<td>7.560 sec</td>
</tr>
<tr>
<td>sub3b</td>
<td>10.440 sec</td>
<td>11.200 sec</td>
</tr>
<tr>
<td>mult1(np)</td>
<td>1133.490 sec</td>
<td>4.690 sec</td>
</tr>
<tr>
<td>mult1a</td>
<td>807.730 sec</td>
<td>4.730 sec</td>
</tr>
<tr>
<td>mult4</td>
<td>∞</td>
<td>17.900 sec</td>
</tr>
<tr>
<td>mult4(np)</td>
<td>∞</td>
<td>13.140 sec</td>
</tr>
<tr>
<td>Program</td>
<td>Depth-First</td>
<td>Memoization</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>plus’4</td>
<td>0.08 sec</td>
<td>0.180 sec</td>
</tr>
<tr>
<td>plus4</td>
<td>0.1 sec</td>
<td>0.430 sec</td>
</tr>
<tr>
<td>plus4(np)</td>
<td>22.770 sec</td>
<td>1.95 sec</td>
</tr>
<tr>
<td>sub’1a</td>
<td>0.050 sec</td>
<td>0.240 sec</td>
</tr>
<tr>
<td>sub1b</td>
<td>0.250 sec</td>
<td>5.020 sec</td>
</tr>
<tr>
<td>sub3b</td>
<td>0.350 sec</td>
<td>8.160 sec</td>
</tr>
<tr>
<td>mult1(np)</td>
<td>1133.490 sec</td>
<td>4.690 sec</td>
</tr>
<tr>
<td>mult1a</td>
<td>0.160 sec</td>
<td>2.900 sec</td>
</tr>
<tr>
<td>mult4</td>
<td>0.250 sec</td>
<td>7.150 sec</td>
</tr>
<tr>
<td>mult4(np)</td>
<td>∞</td>
<td>13.020 sec</td>
</tr>
</tbody>
</table>
Evaluation

- Simple memoization improves performance
- #Entries in table < 300
- #SuspendedGoals < 200
- Quick failure is important for program development
- Overhead of memoization may hurt performance
- Multi-stage strategy delays the reuse of answers
  SCC(strongly connected components)
Type-checker with explicit memoization?

• Investigate special memoization techniques [Davies, Pfenning 00]
• Implementation is non-trivial.
• Proofs are larger.
• Sending and checking proofs takes longer.
• Harder to reason about this implementation
Outline

• LF as a logic programming framework
• Example: Type-system with subtyping
• Basics of tabled higher-order logic programming

• Experimental evaluation:
  1. Refinement type-checking:
     Depth-first vs tabled search
  2. Parsing into higher-order abstract syntax:
     Iterative deepening vs tabled search

• Related Work
• Conclusion and future work
Outline

• LF as a logic programming framework
• Example: Type-system with subtyping
• Basics of tabled higher-order logic programming
• Experimental evaluation:
  1. Refinement type-checking: Depth-first vs tabled search
  2. Parsing into higher-order abstract syntax: Iterative deepening vs tabled search
• Related Work
• Conclusion and future work
Parsing into higher-order abstract syntax

Tokens $T$:

'forall' | 'exist' | 'and' | 'or' | 'imp' | 'not' | '(' | ')' | 'true' | 'false'

Propositions $A$:

atom $P$ | $\neg A$ | $A \& A$ | $A \lor A$ | $A \Rightarrow A$ | true | false |

forall $x.A$ | exists $x.A$ | $(A)$

Precedence $\neg > \& > \lor > \Rightarrow$

Associativity

$\&$, $\lor$: left associative

$\Rightarrow$: right associative
Implementation (idea by D.S. Warren)

% implication -- right associative
fimp: fi Ctx S S’ (P1 => P2)
  <- fo Ctx S (’imp’ ; S1) P1
  <- fi Ctx S1 S’ P2.
ci:  fi Ctx S S’ P
    <- fo Ctx S S’ P.

% disjunction -- left associative
for: fo Ctx S S’ (P1 v P2)
  <- fo Ctx S (’or’ ; S1) P1
  <- fa Ctx S1 S’ P2.
## Iterative Deepening vs Memoization

<table>
<thead>
<tr>
<th>Length of input</th>
<th>Iter. deepening</th>
<th>Memoization</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.020 sec</td>
<td>0.010 sec</td>
</tr>
<tr>
<td>20</td>
<td>1.610 sec</td>
<td>0.260 sec</td>
</tr>
<tr>
<td>32</td>
<td>208.010 sec</td>
<td>2.020 sec</td>
</tr>
<tr>
<td>56</td>
<td>∞</td>
<td>7.980 sec</td>
</tr>
<tr>
<td>107</td>
<td>∞</td>
<td>86.320 sec</td>
</tr>
</tbody>
</table>
Evaluation

• Memoization outperforms iter. deepening
• Iterative deepening requires depth-bound
  - Failure meaningless
  - No decision procedure
• #Entries in table < 1000
• #SuspendedGoals < 1100
• Remarks
  1. Unambiguous parser
  2. Representing tokens as facts
Outline

• LF as a logic programming framework
• Example: Type-system with subtyping
• Basics of tabled higher-order logic programming
• Experimental evaluation:
  1. Refinement type-checking: Depth-first vs tabled search
  2. Parsing into higher-order abstract syntax: Iterative deepening vs tabled search
• Related work
• Conclusion and future work
Higher-order theorem proving

• Tactics and tacticals
  - Isabelle [Paulson86], λProlog[Miller91,Felty93]
  - Need to be rewritten for each specification
  - Requires understanding of prover
  - Proving correctness of tactics often hard

• Memoization-based search
  - User concentrates on specification
  - Generic proof search mechanism
  - Table may contain useful failure information
Deterministic search: an alternative?

• Safe cut: finds exactly one solution
• In general: incomplete
• If there are only ground goals, then deterministic search is complete.
• Less general than memoization-based search
• No overhead
Conclusion

Memoization-based search allows

• generic efficient theorem proving
• execution of more declarative specification
• more efficient execution of implementations
• more flexibility
• small proofs

Memoization has some overhead

• Mixing tabled and non-tabled computation
• Table access
• Table size
Future work

• Higher-order indexing
• Different table strategies
• Incorporate into meta-theorem prover *Twelf* [Schürmann, Pfenning 99]
• Applying tabelling to linear logic programming
Finally ...

Acknowledgements: Frank Pfenning

if you want to find out more:

Demo after workshop

http://www.cs.cmu.edu/~bp
email: bp@cs.cmu.edu
Application: Certified code

Code Producer
- Safety policy
- Generate Certificate

Program

Certificate

Code Consumer
- Safety policy
- Check Certificate
Application: Certified code

- Foundational proof-carrying code: [Appel, Felty 00]
- Proof-carrying authentication: [Felten, Appel 99]
Application: Certified code

- Foundational proof-carrying code: [Appel, Felty 00]
- Proof-carrying authentication: [Felten, Appel 99]
- Proof-checking via bit-strings: [Necula, Rahul 01]
# Runtime, #Entries, #SuspGoals

<table>
<thead>
<tr>
<th>Program</th>
<th>Run-Time</th>
<th>#Entries</th>
<th>#SuspGoals</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus’4</td>
<td>2.330 sec</td>
<td>151</td>
<td>48</td>
</tr>
<tr>
<td>plus4</td>
<td>3.150 sec</td>
<td>171</td>
<td>74</td>
</tr>
<tr>
<td>plus4(np)</td>
<td>1.95 sec</td>
<td>143</td>
<td>56</td>
</tr>
<tr>
<td>sub’1a</td>
<td>0.240 sec</td>
<td>58</td>
<td>11</td>
</tr>
<tr>
<td>sub1b</td>
<td>7.560 sec</td>
<td>252</td>
<td>138</td>
</tr>
<tr>
<td>sub3b</td>
<td>11.200 sec</td>
<td>278</td>
<td>170</td>
</tr>
<tr>
<td>mult1(np)</td>
<td>4.690 sec</td>
<td>217</td>
<td>83</td>
</tr>
<tr>
<td>mult1a</td>
<td>4.730 sec</td>
<td>211</td>
<td>78</td>
</tr>
<tr>
<td>mult4</td>
<td>17.900 sec</td>
<td>298</td>
<td>270</td>
</tr>
<tr>
<td>mult4(np)</td>
<td>13.140 sec</td>
<td>275</td>
<td>194</td>
</tr>
</tbody>
</table>
# Time, #Entries, #SuspGoals

<table>
<thead>
<tr>
<th>Length of input</th>
<th>Memoization</th>
<th>#Entries</th>
<th>#SuspGoals</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.010 sec</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>20</td>
<td>0.260 sec</td>
<td>60</td>
<td>54</td>
</tr>
<tr>
<td>32</td>
<td>2.020 sec</td>
<td>176</td>
<td>197</td>
</tr>
<tr>
<td>56</td>
<td>7.980 sec</td>
<td>371</td>
<td>439</td>
</tr>
<tr>
<td>107</td>
<td>86.320 sec</td>
<td>929</td>
<td>1185</td>
</tr>
</tbody>
</table>