Tabled higher-order logic programming

Thesis Proposal

Brigitte Pientka

Department of Computer Science
Carnegie Mellon University
Outline

- Introduction
- Illustrating example: subtyping
- Tabled higher-order logic programming
  - Tabled logic programming interpreter
  - Object- and meta-level theorem prover
- Thesis work
- Conclusion
Outline

• Introduction
• Illustrating example: subtyping
• Tabled higher-order logic programming
  – Tabled logic programming interpreter
  – Object- and meta-level theorem prover
• Thesis work
• Conclusion
Introduction

- Higher-order logic programming
  Terms: (dependently) typed $\lambda$-calculus
  Clauses: implication, universal quantification
Introduction

- Higher-order logic programming
  Terms: (dependently) typed $\lambda$-calculus
  Clauses: implication, universal quantification

- Meta-language for specifying / implementing logical systems
  proofs about them
Higher-order logic programming
Terms: (dependently) typed λ-calculus
Clauses: implication, universal quantification

Meta-language for specifying / implementing
logical systems (type system, safety logic, congruence closure . . .)
proofs about them
Introduction

- Higher-order logic programming
  Terms: (dependently) typed $\lambda$-calculus
  Clauses: implication, universal quantification

- Meta-language for specifying / implementing logical systems (type system, safety logic, congruence closure . . .)
  proofs about them (correctness, soundness etc.)
Introduction

- Higher-order logic programming
  Terms: (dependently) typed $\lambda$-calculus
  Clauses: implication, universal quantification

- Meta-language for specifying / implementing
  logical systems (type system, safety logic, congruence closure . . .)
  proofs about them (correctness, soundness etc.)

- Approaches: Elf, $\lambda$Prolog, Isabelle
Generic framework for . . .

- Implementing logical systems
- Executing them and generating certificate
- Checking certificate
- Reasoning with and about them
Generic framework for . . .

- Implementing logical systems
  higher-order logic program
- Executing them and generating certificate
- Checking certificate
- Reasoning with and about them
Generic framework for . . .

- Implementing logical systems
  higher-order logic program
- Executing them and generating certificate
  logic programming interpreter Elf
- Checking certificate
- Reasoning with and about them
Generic framework for . . .

- Implementing logical systems
  higher-order logic program
- Executing them and generating certificate
  logic programming interpreter Elf
- Checking certificate
  type checker
- Reasoning with and about them
Generic framework for . . .

- Implementing logical systems
  higher-order logic program
- Executing them and generating certificate
  logic programming interpreter Elf
- Checking certificate
  type checker
- Reasoning with and about them
  object- and meta-level theorem prover Twelf
Generic framework for . . .

• Implementing logical systems
  higher-order logic program

• Executing them and generating certificate
  logic programming interpreter Elf

• Checking certificate
  type checker

• Reasoning with and about them
  object- and meta-level theorem prover Twelf

Reduces the effort required for each logical system
Generic framework for . . .

- Implementing logical systems
  higher-order logic program
- Executing them and generating certificate
  logic programming interpreter Elf
- Checking certificate
  type checker
- Reasoning with and about them
  object- and meta-level theorem prover Twelf

Reduces the effort required for each logical system
Proof search tree

- Search Strategy
  - Depth-first: incomplete, infinite paths
  - Iterative deepening: complete, infinite paths

- Performance: redundant computation
Proof search tree

- Search Strategy
  - Depth-first: incomplete, infinite paths
  - Iterative deepening: complete, infinite paths
- Performance: redundant computation
Proof search tree

- **Search Strategy**
  - Depth-first: incomplete, infinite paths
  - Iterative deepening: complete, infinite paths

- **Performance:** redundant computation
Tabled evaluation for Prolog

- Tabling, memoization, caching, loop detection, magic sets ...
- Eliminate infinite and redundant computation by memoization (Tamaki, Sato)
- Finds all possible answers to a query
- Terminates for programs in a finite domain
- Combines tabled and non-tabled execution
- Very successful: XSB system (Warren et al.)
This talk

1. Extend tabled logic programming to higher-order
2. Demonstrate the use of tabled search to
   • efficiently execute logical systems
   • automate reasoning with and about them.
This talk

1. Extend tabled logic programming to higher-order
2. Demonstrate the use of tabled search to
   • efficiently execute logical systems
     (interpreter using tabled search)
   • automate reasoning with and about them.
1. Extend tabled logic programming to higher-order
2. Demonstrate the use of tabled search to
   • efficiently execute logical systems
     (interpreter using tabled search)
   • automate reasoning with and about them.
     (theorem prover using tabled search)
Outline

• Introduction
• Illustrating example: subtyping
• Tabled higher-order logic programming
  – Tabled logic programming interpreter
  – Object- and meta-level theorem prover
• Thesis work
• Conclusion
Illustrating example: subtyping

Types \( \tau \ ::= \) neg | zero | pos | nat | int
## Illustrating example: subtyping

Types $\tau ::=$ \{ neg, zero, pos, nat, int \}

<table>
<thead>
<tr>
<th>zn</th>
<th>pn</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero $\leq$ nat</td>
<td>pos $\leq$ nat</td>
</tr>
</tbody>
</table>
Illustrating example: subtyping

Types $\tau ::= \text{neg} \mid \text{zero} \mid \text{pos} \mid \text{nat} \mid \text{int}$

$\begin{align*}
\text{zero} & \preceq \text{nat} \\
\text{pos} & \preceq \text{nat} \\
\text{nat} & \preceq \text{int} \\
\text{neg} & \preceq \text{int}
\end{align*}$
Illustrating example: subtyping

Types \( \tau \ ::= \text{neg} | \text{zero} | \text{pos} | \text{nat} | \text{int} \)

\[
\begin{align*}
\text{zn} & \leq \text{nat} \\
\text{pn} & \leq \text{nat} \\
\text{nati} & \leq \text{int} \\
\text{negi} & \leq \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{refl} & \leq T \\
T & \leq R \\
R & \leq S \\
\text{tr} & \leq T
\end{align*}
\]
Subtyping relation in Elf

\text{refl} : \text{sub } T \ T.
\text{tr} : \text{sub } T \ S \\
\quad \leftarrow \text{sub } T \ R \\
\quad \leftarrow \text{sub } R \ S.
\text{zn} : \text{sub zero nat}.
\text{pn} : \text{sub pos nat}.
\text{nati} : \text{sub nat int}.
\text{negi} : \text{sub neg int}.
Compute all supertypes of zero

refl : sub $T$ $T$.
tr : sub $T$ $S$
    ← sub $T$ $R$
    ← sub $R$ $S$.
zn : sub zero nat.
pn : sub pos nat.
nati : sub nat int.
negi : sub neg int.
Subtyping relation in Elf

refl : \( \text{sub } T \ T \).  

\( \text{tr} \) : \( \text{sub } T \ S \)
\( \leftarrow \ \text{sub } T \ R \)
\( \leftarrow \ \text{sub } R \ S \). 

zn : \( \text{sub zero nat} \).

pn : \( \text{sub pos nat} \).

nati : \( \text{sub nat int} \).

negi : \( \text{sub neg int} \).

Compute all supertypes of zero
\( : - ? \ \text{sub zero } T. \)

refl: \( T = \text{zero} \)
Success
Subtyping relation in Elf

compute all supertypes of zero

refl : sub $T T$.

tr : sub $T S$

$\leftarrow$ sub $T R$

$\leftarrow$ sub $R S$.

zn : sub zero nat .

pn : sub pos nat .

nati : sub nat int .

negi : sub neg int .

Compute all supertypes of zero $T$. 

tr : sub zero $R$, sub $R T$. 

Subtyping relation in Elf

refl : sub $T \ T$.  
tr : sub $T \ S$  
    left sub $T \ R$  
    left sub $R \ S$.
zn : sub zero nat .
pn : sub pos nat .
nati : sub nat int .
negi : sub neg int .

Compute all supertypes of zero

: -- ? sub zero $T$.
tr: sub zero $R$, sub $R \ T$.
refl: sub zero $T$
Subtyping relation in Elf

refl : sub $T \ T$.
tr : sub $T \ S$
    $\leftarrow$ sub $T \ R$
    $\leftarrow$ sub $R \ S$.
zn : sub zero nat .
pn : sub pos nat .
nati : sub nat int .
negi : sub neg int .

Compute all supertypes of zero

tr: sub zero $R$, sub $R \ T$.
refl: sub zero $T$
refl: $T = $ zero

Redundant answer
Subtyping relation in Elf

refl : sub \( T T \).

tr : sub \( T S \)

\( \leftarrow \) sub \( T R \)

\( \leftarrow \) sub \( R S \).

zn : sub zero nat .

pn : sub pos nat .

nati : sub nat int .

negi : sub neg int .

Compute all supertypes of zero

\( : - ? \) sub zero \( T \).

tr : sub zero \( R \), sub \( R T \).

refl : sub zero \( T \)

tr : sub zero \( R \), sub \( R T \).
Subtyping relation in Elf

refl :  sub $T \ T$.

tr :  sub $T \ S$

← sub $T \ R$

← sub $R \ S$.

zn :  sub zero nat .

pn :  sub pos nat .

nati :  sub nat int .

negi :  sub neg int .

Compute all supertypes of zero :

: –? sub zero $T$.

tr:  sub zero $R \ , \ sub \ R \ T$.

refl:  sub zero $T$

tr:  sub zero $R \ , \ sub \ R \ T$.

Infinite path
Problem

- Redundant and infinite computation
- Non-termination instead of failure
- Sensitive to clause ordering
- Independent of the actual search strategy
Tabled logic programming

- Eliminate redundant and infinite paths from proof search using memoization
- Table:
  1. Store sub-goals
  2. Store solutions
  3. Retrieve solutions
- Depth-first multi-stage strategy
Tabled computation

%tabled sub.
refl : sub T T.
tr : sub T S
    sub T R
    sub R S.
zn : sub zero nat.
pn : sub pos nat.
nati : sub nat int.
egi : sub neg int.
Compute all supertypes of zero:

```prolog
%tabled sub.

refl : sub T T.

tr : sub T S
    ← sub T R
    ← sub R S.

zn : sub zero nat.

pn : sub pos nat.

nati : sub nat int.

negi : sub neg int.
```

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero T</td>
<td>sub zero T</td>
</tr>
</tbody>
</table>
Compute all supertypes of zero

\( \text{refl} : \quad \text{sub } T \ T. \)
\( \text{tr} : \quad \text{sub } T \ S \)
\( \quad \text{sub } T \ R \)
\( \quad \text{sub } R \ S. \)
\( \text{zn} : \quad \text{sub zero nat} . \)
\( \text{pn} : \quad \text{sub pos nat} . \)
\( \text{nati} : \quad \text{sub nat int} . \)
\( \text{negi} : \quad \text{sub neg int} . \)

**Entry** | **Answer**
---|---
sub zero \( T \)
%tabled sub .
refl : sub \( T \, T \).
tr : sub \( T \, S \)
    ← sub \( T \, R \)
    ← sub \( R \, S \).
zn : sub zero nat .
pn : sub pos nat .
nati : sub nat int .
negi : sub neg int .

Compute all supertypes of zero
: –? sub zero \( T \).
refl : \( T = \text{zero} \)
Add answer to table

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero ( T )</td>
<td>( [\text{zero} / T] )</td>
</tr>
</tbody>
</table>
%tabled sub.
refl : sub \( T \ T \).
tr : sub \( T \ S \)
    \( \leftarrow \) sub \( T \ R \)
    \( \leftarrow \) sub \( R \ S \).
zn : sub zero nat.
pn : sub pos nat.
nati : sub nat int.
negi : sub neg int.

Compute all supertypes of zero
: – ? sub zero \( T \).
tr : sub zero \( R \), sub \( R \ T \).

Variant of previous goal

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero ( T )</td>
<td>[zero /( T )]</td>
</tr>
</tbody>
</table>
%tabled sub.

refl : sub \( T \ T \).

tr : sub \( T \ S \)
    \( \leftarrow \) sub \( T \ R \)
    \( \leftarrow \) sub \( R \ S \).

zn : sub zero nat.

pn : sub pos nat.

nati : sub nat int.

negi : sub neg int.

Compute all supertypes of zero

: \( \leftarrow ? \) sub zero \( T \).

tr : sub zero \( R \), sub \( R \ T \).

Fail and suspend goal

Entry | Answer
----- | -------
sub zero \( T \) | [zero \( / T \)]
Compute all supertypes of zero

refl : sub $T \, T$.
tr : sub $T \, S$
    ← sub $T \, R$
    ← sub $R \, S$.
zn : sub zero nat.

Success!

pn : sub pos nat.
nati : sub nat int.
negi : sub neg int.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero $T$</td>
<td>[zero /$T$]</td>
</tr>
</tbody>
</table>
%tabled sub .
refl : sub $T \, T$.
tr : sub $T \, S$
     ← sub $T \, R$
     ← sub $R \, S$.
zn : sub zero nat .
pn : sub pos nat .
nati : sub nat int .
negi : sub neg int .

Compute all supertypes of zero
:- ? sub zero $T$.
zn : $T = \text{nat}$

Add answer to table

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero $T$</td>
<td>$[\text{zero} / T]$ , $[\text{nat} / T]$</td>
</tr>
</tbody>
</table>
Tabled computation

Compute all supertypes of zero

\[ : - \text{sub zero } T. \]

Add answer to table

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero (T)</td>
<td>([\text{zero} / T], [\text{nat} / T])</td>
</tr>
</tbody>
</table>

First Stage completed!
Tabled computation

Compute all supertypes of zero

:– ? sub zero T.

resume sub zero R, sub R T.

%tabled sub .
refl : sub T T.
tr : sub T S
    ← sub T R
    ← sub R S.
zn : sub zero nat .
pn : sub pos nat .
nati : sub nat int .
negi : sub neg int .

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero T</td>
<td>[zero /T] , [nat /T]</td>
</tr>
</tbody>
</table>
Compute all supertypes of zero

\[ \text{Compute all supertypes of zero} \]

\[ \text{refl} : \quad \text{sub } T \rightarrow T. \]

\[ \text{tr} : \quad \text{sub } T \rightarrow S \]
\[ \quad \text{sub } T \rightarrow R \]
\[ \quad \text{sub } R \rightarrow S. \]

\[ \text{zn} : \quad \text{sub zero nat}. \]

\[ \text{pn} : \quad \text{sub pos nat}. \]

\[ \text{nati} : \quad \text{sub nat int}. \]

\[ \text{negi} : \quad \text{sub neg int}. \]

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero ( T )</td>
<td>([\text{zero} / T]), ([\text{nat} / T])</td>
</tr>
</tbody>
</table>
Tabled computation

Compute all supertypes of zero

%tabled sub.
refl : sub \( T \ T \).
\[ \text{tr : sub } \ T \ S \]
\[ \leftarrow \text{sub } \ T \ R \]
\[ \leftarrow \text{sub } \ R \ S. \]
zn : sub zero nat.

pn : sub pos nat.
nati : sub nat int.
negi : sub neg int.

Compute all supertypes of zero

: - ? sub zero \( T \).

resume sub zero \( R \), sub \( R \ T \).

[ nat / \( R \) ] sub nat \( T \).

Add goal to table

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero ( T )</td>
<td>[ zero / ( T ) ], [ nat / ( T ) ]</td>
</tr>
<tr>
<td>sub nat ( T )</td>
<td></td>
</tr>
</tbody>
</table>
Tabled computation

%tabled sub.
refl : sub T T.
tr : sub T S
  \leftarrow sub T R
  \leftarrow sub R S.
zn : sub zero nat.
pn : sub pos nat.
nati : sub nat int.
negi : sub neg int.

Compute all supertypes of zero
: ? sub zero T.
resume sub zero R , sub R T.
[nat / R] sub nat T
refl T = nat
Success

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero T</td>
<td>[zero / T] , [nat / T]</td>
</tr>
<tr>
<td>sub nat T</td>
<td></td>
</tr>
</tbody>
</table>
Compute all supertypes of zero

refl : sub $T \ T$.
tr : sub $T \ S$
    ← sub $T \ R$
    ← sub $R \ S$.
zn : sub zero nat .
pn : sub pos nat .
nati : sub nat int .
negi : sub neg int .

Resume sub zero $R$ , sub $R \ T$.

Add answer to table

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero $T$</td>
<td>[zero /$T$] , [nat /$T$]</td>
</tr>
<tr>
<td>sub nat $T$</td>
<td>[nat /$T$]</td>
</tr>
</tbody>
</table>
Completely higher-order logic programming – p. 13/40

Tabled computation

%tabled sub.
refl : sub T T.
tr : sub T S
   ← sub T R
   ← sub R S.
zn : sub zero nat.
pn : sub pos nat.
nati : sub nat int.
egi : sub neg int.

Compute all supertypes of zero
:- ? sub zero T.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub zero T</td>
<td>[zero / T] , [nat / T] , [int / T]</td>
</tr>
<tr>
<td>sub nat T</td>
<td>[nat / T] , [int / T]</td>
</tr>
<tr>
<td>sub int T</td>
<td>[int / T]</td>
</tr>
</tbody>
</table>
• When to suspend goals?
Strategy

- When to suspend goals?
- When to retrieve answers?
When to suspend goals?
When to retrieve answers?
How to retrieve answers (order)?
Strategy

• When to suspend goals?
• When to retrieve answers?
• How to retrieve answers (order)?
• What is the retrieval condition?
  – Variant
  – Subsumption
Strategy

• When to suspend goals?
• When to retrieve answers?
• How to retrieve answers (order)?
• What is the retrieval condition?
  – Variant
  – Subsumption

Multi-stage strategy:
  only re-use answers from previous stages
Advantages

- Translating inference rules to logic program is straightforward.
- Programs have better complexities.
- Order of clauses is less important.
- Computation will terminate for finite domain.
- We find all answers to a query.
- We can dis-prove more conjectures.
- Table contains useful debugging information.
Trade-off

Price to pay:

- More complicated semantics
- Overhead caused by memoization
Trade-off

Price to pay:
- More complicated semantics
- Overhead caused by memoization

Solution:
- Combine tabled and non-tabled proof search
- Term indexing:
  1. Make table access efficient
  2. Make storage space small
First-order tabled logic programming

- Tabled logic programming
  - atomic subgoals
  - untyped first-order terms
- Procedural descriptions of tabling
  - SLD resolution with memoization (Tamaki, Sato)
  - SLG resolution (Warren, Chen)
- Term indexing (I.V.Ramakrishnan, Sekar, Voronkov)
  discrimination tries, substitution trees, path indexing
Outline

- Introduction
- Illustrating example: subtyping
- Tabled higher-order logic programming
  - Tabled logic programming interpreter
  - Object- and meta-level theorem prover
- Thesis work
- Conclusion
Outline

- Introduction
- Illustrating example: subtyping
- Tabled higher-order logic programming
  - Tabled logic programming interpreter
  - Object- and meta-level theorem prover
- Thesis work
- Conclusion
Typing rules

Mini ML  \( e ::= n(e) \mid z \mid s(e) \mid \text{app } e_1 \ e_2 \mid \text{lam } x.e \mid \text{letn } u = e_1 \ \text{in} \ e_2 \)

\[
\begin{align*}
\Gamma \vdash e : \tau' & \quad \tau' \leq \tau \quad \text{tp-sub} \\
\Gamma \vdash e : \tau & \\
\Gamma, x : \tau_1 \vdash e : \tau_2 & \quad \text{tp-lam} \\
\Gamma \vdash \text{lam } x.e : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash [e_1/u]e_2 : \tau & \quad \text{tp-letn} \\
\Gamma \vdash \text{letn } u = e_1 \ \text{in} \ e_2 : \tau
\end{align*}
\]
Type Checker in Elf

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

\[
\text{tp-lam : of (lam ([x] E x)) (T_1 \Rightarrow T_2)}
\]

\[
\leftarrow (\{x\} \text{of } x \quad T_1 \rightarrow \text{ of } (E \quad x) \quad T_2).
\]

\[
\text{tp-letn : of (letn E_1 ([u] E_2 u)) T}
\]

\[
\leftarrow \text{ of } \quad E_1 \quad T_1 \\
\leftarrow \text{ of } \quad (E_2 \quad E_1) \quad T.
\]
Tabled computation (higher-order)

:– ? of (lam ([x] x)) T

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>of (lam ([x] x)) T</td>
<td></td>
</tr>
</tbody>
</table>
Tabled computation (higher-order)

\[ \text{tp-sub: of (lam } ([x] x)) R, \text{ sub } R T. \]

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>of (lam ([x] x)) T</td>
<td></td>
</tr>
</tbody>
</table>
Tabled computation (higher-order)

:- ? of (lam ([x] x)) T

\text{tp-sub}: \text{of} \ (\text{lam} \ (\text{[x] x})) \ R, \ \text{sub} \ R \ T.

Variant of previous goal

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>of (lam ([x] x)) T</td>
<td></td>
</tr>
</tbody>
</table>
Tabled computation (higher-order)

\[ \text{tp-sub: of (lam ([x] x)) R, sub } R T. \]

Fail and suspend

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>of (lam ([x] x)) T</td>
<td></td>
</tr>
</tbody>
</table>
Tabled computation (higher-order)

:–? of (lam ([x] x)) T

tp-lam: u : of x T₁ ⊢ of x T₂

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>of (lam ([x] x)) T</td>
<td></td>
</tr>
</tbody>
</table>
### Tabled computation (higher-order)

:− ? of (lam ([x] x)) T

\[ \text{tp-lam: } u : \text{of } x \ T_1 \vdash \text{of } x \ T_2 \]

Add goal to table

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>of (lam ([x] x)) T</td>
<td>U : of x T_1 \vdash of x T_2</td>
</tr>
</tbody>
</table>
Tabled computation (higher-order)

: – ? of (lam ([x] x)) T

\[ u : of \ x \ T_1 \vdash of \ x \ T_2 \]

\[ u : T_1 = P \ , \ T_2 = P \ , \ T = (P \Rightarrow P) \]

Success

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>of (lam ([x] x)) T</td>
<td></td>
</tr>
<tr>
<td>u : of x T_1 \vdash of x T_2</td>
<td></td>
</tr>
</tbody>
</table>
Tabled computation (higher-order)

:− ? of (lam ([x] x)) T

tp-lam: u : of x T₁ ⊨ of x T₂

u: T₁ = P , T₂ = P , T = (P ⇒ P)

Add answers to table

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>of (lam ([x] x)) T</td>
<td>[(P ⇒ P)/T]</td>
</tr>
<tr>
<td>u : of x T₁ ⊨ of x T₂</td>
<td>[P/T₁ , P/T₂]</td>
</tr>
</tbody>
</table>
### Tabled computation (higher-order)

\[ : \neq \text{of} \ (\text{lam} \ (\ldbracket x \rddbracket \ x)) \ T \]

**tp-lam:** \( u : \text{of} \ x \ T_1 \vdash \text{of} \ x \ T_2 \)

**tp-sub:** \( u : \text{of} \ x \ T_1 \vdash \text{of} \ x \ R \), sub \( R \ T_2 \)

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u : \text{of} \ x \ T_1 \vdash \text{of} \ x \ T_2 )</td>
<td>( [(P \Rightarrow P)/T] )</td>
</tr>
<tr>
<td></td>
<td>([P/T_1 \ , \ P/T_2] )</td>
</tr>
</tbody>
</table>
### Tabled computation (higher-order)

\[
\text{tp-lam: } u : \text{of } x \ T_1 \vdash \text{of } x \ T_2
\]

\[
\text{tp-sub: } u : \text{of } x \ T_1 \vdash \text{of } x \ R , \text{ sub } R \ T_2
\]

Variant of previous goal

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{of (lam } ([x] \ x)) \ T )</td>
<td>([(P \Rightarrow P)/T])</td>
</tr>
<tr>
<td>( u : \text{of } x \ T_1 \vdash \text{of } x \ T_2 )</td>
<td>([P/T_1 , P/T_2])</td>
</tr>
</tbody>
</table>
Tabled computation (higher-order)

\[ : \neg ? \text{ of } (\text{lam } ([x] x)) \ T \]

\begin{align*}
\text{tp-lam: } u : & \text{ of } x \ T_1 \vdash \text{ of } x \ T_2 \\
\text{tp-sub: } u : & \text{ of } x \ T_1 \vdash \text{ of } x \ R , \ \text{sub } R \ T_2
\end{align*}

Suspend and fail

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>of (lam ([x] x)) \ T</td>
<td>[(P \Rightarrow P)/T]</td>
</tr>
<tr>
<td>\mbox{u : of } x \ T_1 \vdash \text{ of } x \ T_2</td>
<td>[P/T_1, P/T_2]</td>
</tr>
</tbody>
</table>
Tabled computation (higher-order)

:- ? of (lam ([x] x)) T

First stage is completed

<table>
<thead>
<tr>
<th>Entry</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>of (lam ([x] x)) T</td>
<td>[[(P \Rightarrow P)/T]]</td>
</tr>
<tr>
<td>(u : \text{of } x \ T_1 \leftarrow \text{of } x \ T_2)</td>
<td>([P/T_1, P/T_2]]</td>
</tr>
</tbody>
</table>
Challenges

- Store goals together with context: \( \Gamma \vdash a \)
- Redesign table operations: goal \((\Gamma \vdash a) \in \text{Table}\)
- Context dependencies
  - e.g. \( u : \text{of}\ x\ T_1 \vdash \text{sub}\ R\ T_2, \)
  - \( \vdash \text{sub}\ S\ T \)
- Type dependencies
  - e.g. \( u : \text{of}\ x\ T_1 \vdash \text{of}\ x\ (R\ x\ u), \)
  - \( u : \text{of}\ x\ T_1 \vdash \text{of}\ x\ R \)
- Indexing for higher-order terms
Outline

- Introduction
- Illustrating example: subtyping
- Tabled higher-order logic programming
  - Tabled logic programming interpreter
  - Object- and meta-level theorem prover
- Thesis work
- Conclusion
Theorem proving

• Object-level
  – Prove derived rules
  – Lemma: $D : \text{sub zero int}$.  
  – Derive from program clauses + lemmas

• Meta-level
  – Prove theorems about the logical system
  – Theorem: If $D : \text{of } e \tau$ and $E : \text{eval } e \nu$ then $\mathcal{F} : \text{of } \nu \tau$.
  – Proofs by structural induction and case analysis
Current approaches

• λProlog(Felty, Miller), Isabelle(Paulson): based on tactics

• Twelf(Schürmann, Pfenning) : based on higher-order logic programming iterative deepening with bound
Meta-level search

- Clauses: program, lemmas, proof assumptions
- Proof obligation (query): derive from clauses
- If we cannot derive the query from the clauses,
  1. Refine proof assumptions: case split (choice!)
  2. Generate induction hypothesis
  3. Try again
Meta-Search

1. iteration

2. iteration

3. iteration

Failure

Success
Redundant computation

- Object-level search
- Across failed proof attempts
- Across branches
- Across different parallel proof attempts

Meta-level proof tree
Benefits of tabled meta-level search

- Redundancy elimination during object-level search
- Preservation of partial results across cases and iterations
- Detection of unprovable branches
- Faster failure
- Proving different case split in parallel
- Detection of redundant case splits (e.g. split a and then split b, split b and then split a)
Outline

• Introduction
• Illustrating example: subtyping
• Tabled higher-order logic programming
  – Tabled logic programming interpreter
  – Object- and meta-level theorem prover
• Thesis work
• Conclusion
Tabled higher-order logic programming allows us to

- efficiently execute logical systems

- automate reasoning with and about them.
Tabled higher-order logic programming allows us to

- efficiently execute logical systems (interpreter using tabled search)
- automate reasoning with and about them.
Tabled higher-order logic programming allows us to

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

- Proof-theoretical characterization: Soundness of interpreter
- Design of efficient implementation techniques
  1. Higher-order term indexing
  2. Context handling
- Implementation and Validation
  1. Logic programming
  2. Object and meta-level theorem proving
### Examples: interpreter - 1

<table>
<thead>
<tr>
<th>subtyping1</th>
<th>Elf</th>
<th>variant</th>
<th>subsumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>zsuper</td>
<td>∞</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>casez1</td>
<td>∞</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>disprove</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zerop</td>
<td>∞</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>casez2</td>
<td>∞</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>subtyping</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tid</td>
<td>∞</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>sarrow</td>
<td>∞</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
**Examples: interpreter - 2**

Warning: table everything; no indexing

<table>
<thead>
<tr>
<th>Term rewrite $\lambda$-calculus:</th>
<th>Elf variant</th>
<th>subsumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>tid5</td>
<td>no</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>comb</td>
<td>no</td>
<td>$\checkmark$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Refinement types:</th>
</tr>
</thead>
<tbody>
<tr>
<td>shiftl</td>
</tr>
<tr>
<td>inc</td>
</tr>
<tr>
<td>plus</td>
</tr>
<tr>
<td>plus’</td>
</tr>
</tbody>
</table>
### Object-level reasoning - 3

**Warning:** table everything; no indexing

<table>
<thead>
<tr>
<th></th>
<th>Spass</th>
<th>Twelf</th>
<th>variant</th>
<th>subsumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>conversions λ-calculus:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tid5</td>
<td>no</td>
<td>√</td>
<td>na</td>
<td></td>
</tr>
<tr>
<td>comb</td>
<td>no</td>
<td>√</td>
<td>na</td>
<td></td>
</tr>
<tr>
<td>Cartesian closed categories:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l1</td>
<td>no</td>
<td>no</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>l2</td>
<td>no</td>
<td>no</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>l3</td>
<td>no</td>
<td>no</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Other examples

Logical systems:

- Natural deduction calculi (NK, NJ)
- Decision procedures (e.g. congruence closure algorithms)
- Parsing grammars

Examples for meta-reasoning:

- Soundness of Kolmogoroff translation between NK and NJ
- Translation between CCC and λ-calculus
Outline

• Introduction
• Illustrating example: subtyping
• Tabled higher-order logic programming
  – Tabled logic programming interpreter
  – Object- and meta-level theorem prover
• Thesis work
• Conclusion
Contributions

- Extension of tabling to higher-order setting
  1. Terms: dependently typed $\lambda$-calculus
  2. Table: store goals with a context
- Application of tabled search to
  1. higher-order logic programming
  2. object- and meta-level theorem proving
- Proof-theoretical characterization of tabled search
- Implementation of a prototype
Near Future

- Soundness of the interpreter
- Indexing for higher-order terms
- Redesign of the meta-theorem prover
Related Work

Proof-theoretical characterization

- Uniform proofs (Miller, Nadathur, Pfenning, Scedrov)
- Proof Irrelevance (Pfenning)

Certificates:

- Justifiers: XSB (Roychoudhury, I.V.Ramakrishnan)
- Bit-strings: variant of PCC (Necula, Rahul)
- Proof terms: Elf, Twelf (Schürmann, Pfenning)