Delimited Continuations for Prolog

Tom Schrijvers

Kiel Declarative Programming Days 2013
Motivation
Delimited Continuations

- from Functional Programming
  - Felleisen POPL’88
  - Danvy & Filinski LFP’90
- greatly underused and underappreciated
Prolog lacks infrastructure to capture control patterns
Modular Search Heuristics

PADL 2013 invited talk
Existing Solutions

- Individual Language Extensions
- Awkward Assert/Retract scoping
- Meta-Programming / Program Transformation
  - DCGs
  - Extended DCGs
  - Structured State threading
  - Logical Loops
  - ....
Delimited Continuations
Delimited Continuations for Prolog

TOM SCHRIJVERS
Ghent University, Belgium
(e-mail: tom.schrijvers@ugent.be)

BART DEMOEN
KU Leuven, Belgium
(e-mail: bart.demoen@cs.kuleuven.be)

BENOIT DESOUTER
Ghent University, Belgium
(e-mail: benoit.desouter@ugent.be)

JAN WIELEMAKER
University of Amsterdam, The Netherlands
(e-mail: jan@swi-prolog.org)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Delimited continuations are a famous control primitive that originates in the functional programming world. It allows the programmer to suspend and capture the remaining part of a computation in order to resume it later. We put a new Prolog-compatible face on this primitive and specify its semantics by means of a meta-interpreter. Moreover, we establish the power of delimited continuations in Prolog with several example definitions of high-level language features. Finally, we show how to easily and effectively add delimited continuations support to the WAM.

KEYWORDS: delimited continuations, Prolog

1 Introduction

As a programming language Prolog is very lean. Essentially it consists of Horn clauses extended with mostly simple built-in predicates. While this minimality has several advantages, the lack of infrastructure to capture and facilitate common programming patterns can be quite frustrating. Fortunately, programmers can mitigate the tedious drudgery of encoding frequent programming patterns by automating them by means of Prolog’s rich meta-programming and program transformation facilities. Well-known examples of these are definite clause grammars (DCGs), extended DCGs (Roy 1989), Ciao Prolog’s structured state threading (Ivanovic et al. 2009) and logical loops (Schimpf 2002).

However, non-local program transformations are not ideal for defining new language features for several reasons. Firstly, the effort of defining a transformation is proportional to the number of features in the language – the more features are added, the harder it becomes. Secondly, program transformations are fragile with respect to language evolution: they require amendments when other features are added to the language. Thirdly, when the new feature is introduced in existing
Many Uses

- Modular Search Heuristics
- Implicit Environment
- Exceptions
- Tabling
- Definite Clause Grammars
- Implicit State
- Ciao Prolog's Signal Handling
- Logging
- Iterators
- Iteratees
- Coroutines
- Transducers

Many Uses
Delimited Continuations

- much **easier** than you think
- many **applications**
- just **what Prolog needs**
- **for your language of choice too!**
This Talk

Semantics

Applications

Implementation
Semantics

What are they?
Two New Primitives

reset(\text{Goal,Continuation,Term})

shift(\text{Term})
Plain Reset

main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    writeln(b).
Plain Reset

```
main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    writeln(b).

?- main.
```

"a
b
c"
Plain Reset

main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    writeln(b).

?- main.
Plain Reset

\[
\text{main} :- \\
\quad \text{reset}(p, \_\_\_\_), \\
\quad \text{writeln}(c).
\]

\[
p :- \\
\quad \text{writeln}(a), \\
\quad \text{writeln}(b).
\]

?- main.
Plain Reset

main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    writeln(b).

?- main.
a
b
c
Plain Reset

main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    writeln(b).

?- main.
a
b
Plain Reset

main :-
  reset(p,_,_),
  writeln(c).

p :-
  writeln(a),
  writeln(b).

?- main.
  a
  b
  c
Aborting

main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    shift(_),
    writeln(b).

?- main.
Aborting

```prolog
main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    shift(_),
    writeln(b).

?- main.
```
Abort

\[\text{main} : - \]
\[\text{\hspace{1em} reset}(p,\_,\_,\_),\]
\[\text{\hspace{1em} writeln}(c).\]

\[\text{p} : - \]
\[\text{\hspace{1em} writeln}(a),\]
\[\text{\hspace{1em} shift}(\_),\]
\[\text{\hspace{1em} writeln}(b).\]

?- main.
Aborting

main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    shift(_),
    writeln(b).

?- main.
main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    shift(_),
    writeln(b).

?- main.
Abort

main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    shift(_),
    writeln(b).

?- main.
a
main :-
    reset(p,_,_),
    writeln(c).

p :-
    writeln(a),
    shift(_),
    writeln(b).

?- main.
a
c
Term Passing

main :-
    reset(p,_,X),
    writeln(X),
    writeln(c).

p :-
    writeln(a),
    shift(hello),
    writeln(b).

?- main.
Term Passing

```
main :-
    reset(p,_,X),
    writeln(X),
    writeln(c).

p :-
    writeln(a),
    shift(hello),
    writeln(b).
```

?- main.
a
hello
c
**Term Passing**

```prolog
main :-
    reset(p,_,X),
    writeln(X),
    writeln(c).

p :-
    writeln(a),
    shift(hello),
    writeln(b).
```

?- main.

```
a
hello
c
```
Term Passing

main :-
  reset(p, __, X),
  writeln(X),
  writeln(c).

p :-
  writeln(a),
  shift(hello),
  writeln(b).

?- main.
a
hello

c

add transitions
Continuation

main :-
    reset(p,Cont,__),
    writeln(c),
    call(Cont).

p :-
    writeln(a),
    shift(_),
    writeln(b).

?- main.
a
c
b
Repeated Call

main :-
    reset(p,Cont,_),
    writeln(c),
    call(Cont),
    call(Cont).

p :-
    writeln(a),
    shift(_),
    writeln(b).

?- main.
Repeated Call

main :-
  reset(p,Cont,_),
  writeln(c),
  call(Cont),
  call(Cont).

p :-
  writeln(a),
  shift(_),
  writeln(b).

?- main.
a
c
b
b

add transitions
Repeated Call

main :-
  reset(p,Cont,_,),
  writeln(c),
  call(Cont),
  call(Cont).

p :-
  writeln(a),
  shift(_),
  writeln(b).

?- main.
  a
  c
Repeated Call

main :-
  reset(p,Cont,__),
  writeln(c),
  call(Cont),
  call(Cont).

p :-
  writeln(a),
  shift(_),
  writeln(b).

?- main.
a
c
b
Repeated Call

main :-
    reset(p,Cont,_),
    writeln(c),
    call(Cont),
    call(Cont).

p :-
    writeln(a),
    shift(_),
    writeln(b).

?- main.
?- \text{reset}(true, \text{Cont}, \text{Term}).
No Shift

?- reset(true, Cont, Term).
Cont = 0,
Term = 0.
No Reset

?- shift(x).
No Reset

?- shift(x).
ERROR: Unhandled shift: x
Backtracking

main :-
  reset(p,Cont,__),
  writeln(c),
  call(Cont).

p :-
  shift(__),
  writeln(a).

p :-
  shift(__),
  writeln(b).

?- main.
Backtracking

main :-
    reset(p,Cont,_),
    writeln(c),
    call(Cont).

p :-
    shift(_),
    writeln(a).

?- main.
c

p :-
    shift(_),
    writeln(b).
Backtracking

\[
\begin{align*}
\text{main} & : - \\
& \quad \text{reset}(p, \text{Cont}, _), \\
& \quad \text{writeln}(c), \\
& \quad \text{call}(\text{Cont}). \\
\end{align*}
\]

\[
\begin{align*}
p & : - \\
& \quad \text{shift}(_), \\
& \quad \text{writeln}(a). \\
\end{align*}
\]

\[
\begin{align*}
p & : - \\
& \quad \text{shift}(_), \\
& \quad \text{writeln}(b). \\
\end{align*}
\]

?- main.

\[
\begin{align*}
c \\
a
\end{align*}
\]
Backtracking

main :-
  reset(p,Cont,_),
  writeln(c),
  call(Cont).

p :-
  shift(_),
  writeln(a).

p :-
  shift(_),
  writeln(b).

?- main.
c
a ;
c
b
Backtracking

main :-
    reset(p,Cont,__), writeln(c),
    call(Cont).

p :-
    shift(_), writeln(a).

p :-
    shift(_), writeln(b).

?- main.
c a ;
c
Backtracking

main :-
    reset(p,Cont,_),
    writeln(c),
    call(Cont).

p :-
    shift(_),
    writeln(a).

p :-
    shift(_),
    writeln(b).

?- main.
c
a ;
c
b
This Talk

Semantics

Applications

Implementation
Applications

What are they useful for?
Definite Clause Grammars

Implicit State
Definite Clause Grammars
Definite Clause Grammars

ab --> [].
ab --> [a], [b], ab.

?- phrase(ab,[a,b,a,b],[[]]).
true.
Program Transformation

\[
\begin{align*}
\text{ab} & \rightarrow []. \\
\text{ab} & \rightarrow [a], [b], \text{ab}.
\end{align*}
\]

\text{static program transformation}

\[
\begin{align*}
\text{ab}(L, L). \\
\text{ab}([a, b|L], T) :\text{-} \text{ab}(L, T).
\end{align*}
\]
Program Transformation

\[
\text{ab} \rightarrow [].
\]
\[
\text{ab} \rightarrow [a], [b], \text{ab}.
\]
\[
\text{ab(L,L)}.
\]
\[
\text{ab}([\text{a,b}|L],T) :- \text{ab} (L,T).
\]
\[
\text{phrase}(G,L,T) :- \text{call}(G,L,T).
\]

static program transformation
Disadvantages of Approach

- **Special Syntax**: a lot of refactoring effort required to introduce in large programs
- **Incompatibility**
  - existing control operations like catch/throw
  - not robust wrt syntactic extensions
  - potentially quadratic effort to make all syntax extensions compatible
Delimited Continuations to the Rescue!
Effect Handlers

- McBride: Frank language
- Pretnar & Bauer: Eff language
- Kammar et al. ICFP’13
- Brady ICFP’13
- Kiselyov et al. Haskell’13
Effect Handler

Approach

- Command Syntax
- Command Semantics = Handler
DCGs

c/1 phrase/3

ab.
ab :- c(a), c(b), ab.

?- phrase(ab,[a,b,a,b],[[]]).
true.
DCGs

\texttt{c/1}\hspace{1cm} \texttt{phrase/3}

\texttt{ab.}
\texttt{ab :- c(a), c(b), ab.}

\texttt{?- phrase(ab, [a, b, a, b], []).}
\texttt{true.}
DCGs

command

c/1

phrase/3

ab.
ab :- c(a), c(b), ab.

?- phrase(ab,[a,b,a,b],[[]]).
true.
DCGs

command

\text{c/1}

phrase/3

ab.

\text{ab} :- \text{c(a)}, \text{c(b)}, \text{ab}.

\text{example code}

?- \text{phrase(ab,[a,b,a,b],[[]]). true.}
DCGs

**c/1**

\[ \text{ab.} \]
\[ \text{ab} : - \text{c} (\text{a}), \text{c} (\text{b}), \text{ab}. \]

**phrase/3**

?- phrase(ab,[a,b,a,b],[[]]).
true.
Syntax

c(X) :- shift(c(X)).
Semantics: Handler

\[
\text{phrase}(G, L, T) :-
\quad \text{reset}(G, \text{Cont}, \text{Command}),
\quad \text{if Command = c}(X) \rightarrow
\quad \text{L} = [X|NL],
\quad \text{phrase}(\text{Cont}, NL, T)
\; ;
\quad \text{L} = T
\).
Implicit State

code:

get/1, put/1  \hspace{1cm} \text{runState}/3

inc :-
  get(S),
  NS is S + 1,
  put(NS).

?- runState((inc,inc),0,S).
S = 2.
Command Syntax

get(S) :- shift(get(S)).
put(S) :- shift(put(S)).
ExceptionHandler

runState(G, Sin, Sout) :-
    reset(G, Cont, Command),
    ( Command = \texttt{get}(S) ->
        S = Sin,
        runState(Cont, Sin, Sout)
    ; Command = \texttt{put}(S) ->
        runState(Cont, S, Sout)
    ;
    Sout = Sin
).


Alternative Semantics
Implicit State

get/1, put/1  traceState/4

inc :-
    get(S),
    NS is S + 1,
    put(NS).

?- traceState((inc,inc),0,S,T).
   T = [0,1], S = 2.
traceState(G,Sin,Sout,Trace) :-
    reset(G,Cont,Command),
    ( Command = get(S) ->
        S = Sin,
        traceState(Cont,Sin,Sout,Trace)
    ; Command = put(S) ->
        Trace = [Sin|NTrace],
        traceState(Cont,S,Sout,NTrace)
    ;
        Trace = [], Sout = Sin
    ).
Compositional Handlers
Example

\[
\text{inc} :\!- \text{ get}(S), \text{ NS is } S + 1, \text{ put}(\text{NS}).
\]

\[
\text{ab}.
\]

\[
\text{ab} :\!- \text{ c}(a), \text{ c}(b), \text{ inc}, \text{ ab}.
\]

?- \text{runState}(
  \text{phrase}(\text{ab}, [a,b,a,b], [[]]),
  \emptyset, S).

S = 2.
Example

\texttt{inc :- get(S), NS is S + 1, put(NS).}
\texttt{ab.}
\texttt{ab :- c(a), c(b), inc, ab.}

\texttt{?- phrase(}
\texttt{runState(ab,0,S),}
\texttt{[a,b,a,b],[]).}
\texttt{S = 2.}
Compositional Handler

\[
\text{phrase}(G,L,T) :- \\
\quad \text{reset}(G,\text{Cont},\text{Command}), \\
\quad ( \text{Command} = \text{c}(X) \rightarrow \\
\quad \quad L = [X|NL], \\
\quad \quad \text{phrase}(\text{Cont},NL,T) \\
\quad ; \text{Command} = 0 \rightarrow \\
\quad \quad L = T \\
\quad ; \\
\quad \quad \text{shift}(\text{Command}), \\
\quad \quad \text{phrase}(\text{Cont},L,T) \\
\).
Many Uses

- Definite Clause Grammars
- Implicit State
- Implicit Environment
- Exceptions
- Ciao Prolog’s Signal Handling
- Logging
- Iterators
- Iteratees
- Coroutines
- Transducers
Implementation

How to implement them?
Meta-Interpreter
Vanilla Interpreter

eval(true) :- !.
eval((G1,G2)) :- !,
    eval(G1),
    eval(G2).
eval(Goal) :-
    clause(Goal,Body),
    eval(Body).
D.C. Interpreter

\texttt{eval(+Goal,-Status)}

\textbf{Status:}
\begin{itemize}
  \item \texttt{ok}
  \item \texttt{shift(Term,Cont)}
\end{itemize}
D.C. Interpreter

eval(shift(Term),Status) :- !,
    Status = shift(Term,true).
eval(reset(G,Cont,Term),Status) :- !,
    Status = ok,
    eval(G,S),
    ( S == ok ->
    Cont = 0, Term = 0
    ;
    S = shift(Term,Cont) ).
D.C. Interpreter

eval(true,Status) :- !, Status = ok.
eval((G1,G2),Status) :- !,
eval(G1,S1), ( S1 == ok ->
    eval(G2,Status)
; S1 = shift(Term,Cont) ->
    NCont = (Cont,G2),
    Status = shift(Term,NCont)
).

D.C. Interpreter

eval(Goal, Status) :- !, clause(Goal, Body), eval(Body, Status).
Meta-Interpreter

- easy to define and understand
- executable specification
- does not scale well to other features
- poor performance
WAM
Warren Abstract Machine
Catch & Throw

Goal :- ... throw(Term) ...  
?- catch(Goal,Ball,Handler), ...

1. unify a copy of Term with Ball 
2. unwind environment & choice point stacks up to catch/3 
3. Handler is called before control goes to ...
Reset & Shift

Goal :- ... $\text{shift}$(Term) ...

?- $\text{reset}$(Goal,Cont,Ball), ...

1. unify Term with Ball
2. leave the stacks intact
3. unify Cont with a copy of the environment up to reset/3
4. Control goes to ...
Four Issues

1. up to reset/3
2. how to copy (a delimited part of) the environment stack
3. how to use this delimited continuation
4. fineprint
Up to reset/3

same principle as catch/throw
Up to reset/3
Continuation Term

Environment Stack

Heap

\( a(X) : - b, c(X,Y), \text{shift}(1), d(Y). \)

\$cont\$(ContCP,[X,Y])
Reified Environment

Environment Stack

Heap

$\texttt{cont}(\text{ContCP},[X,Y])$
Environment Chain

Environment Stack

Heap

\[
[\text{cont}(\text{CP1}, \text{Vars1}), \text{cont}(\text{CP2}, \text{Vars2}), \text{cont}(\text{CP3}, \text{Vars3})]
\]
Callable Continuation Term

Cont = call_continuation(  
  [$cont$(CP1,Vars1),  
  $cont$(CP2,Vars2),  
  $cont$(CP3,Vars3)])

call_continuation([]).
call_continuation([[Chunk|Chunks]]) :-
call_chunk(Chunk),  
call_continuation(Chunks).
Performance

❖ Not the main focus
❖ Pretty Decent
Shift Runtime (ms)

```
main :- reset(p1,_,_).
dummy.
p1 :- p2, dummy.
p2 :- p3, dummy.
... p5000 :- shift(_), dummy.
```
# Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Native</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>1,965</td>
<td>164</td>
</tr>
<tr>
<td>10,000</td>
<td>3,950</td>
<td>328</td>
</tr>
<tr>
<td>20,000</td>
<td>8,388</td>
<td>664</td>
</tr>
</tbody>
</table>
Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Native hProlog</th>
<th>Native SWI-Prolog</th>
<th>Transformed hProlog</th>
<th>Transformed SWI-Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>1,965</td>
<td>1,965</td>
<td>1,965</td>
<td>1,965</td>
</tr>
<tr>
<td>10,000</td>
<td>3,950</td>
<td>3,950</td>
<td>3,950</td>
<td>3,950</td>
</tr>
<tr>
<td>20,000</td>
<td>8,388</td>
<td>8,388</td>
<td>8,388</td>
<td>8,388</td>
</tr>
</tbody>
</table>

specialization of meta-interpreter
## Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Transformed</th>
<th>hProlog</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>1,965</td>
<td>164</td>
</tr>
<tr>
<td>10,000</td>
<td>3,950</td>
<td>328</td>
</tr>
<tr>
<td>20,000</td>
<td>8,388</td>
<td>664</td>
</tr>
</tbody>
</table>

*specialization of meta-interpreter*
## Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>hProlog</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>164</td>
</tr>
<tr>
<td>10,000</td>
<td>328</td>
</tr>
<tr>
<td>20,000</td>
<td>664</td>
</tr>
</tbody>
</table>
## Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Native</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td></td>
<td>164</td>
</tr>
<tr>
<td>10,000</td>
<td></td>
<td>328</td>
</tr>
<tr>
<td>20,000</td>
<td></td>
<td>664</td>
</tr>
</tbody>
</table>
## Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Native</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>hProlog</td>
<td>hProlog</td>
</tr>
<tr>
<td>5,000</td>
<td></td>
<td>164</td>
</tr>
<tr>
<td>10,000</td>
<td></td>
<td>328</td>
</tr>
<tr>
<td>20,000</td>
<td></td>
<td>664</td>
</tr>
</tbody>
</table>

*WAM architecture*
## Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Native hProlog</th>
<th>Transformed hProlog</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>64</td>
<td>164</td>
</tr>
<tr>
<td>10,000</td>
<td>128</td>
<td>328</td>
</tr>
<tr>
<td>20,000</td>
<td>268</td>
<td>664</td>
</tr>
</tbody>
</table>
## Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Native</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hProlog</td>
<td>64</td>
<td>164</td>
</tr>
<tr>
<td>5,000</td>
<td>128</td>
<td>328</td>
</tr>
<tr>
<td>20,000</td>
<td>268</td>
<td>664</td>
</tr>
</tbody>
</table>
## Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>hProlog</th>
<th>Native</th>
<th>hProlog</th>
<th>Transformed</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>64</td>
<td>hProlog</td>
<td>164</td>
<td>505</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>128</td>
<td>hProlog</td>
<td>328</td>
<td>1,028</td>
<td></td>
</tr>
<tr>
<td>20,000</td>
<td>268</td>
<td>hProlog</td>
<td>664</td>
<td>2,037</td>
<td></td>
</tr>
</tbody>
</table>
# Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Native hProlog</th>
<th>Native SWI-Prolog</th>
<th>Transformed SWI-Prolog</th>
<th>ZIP architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>64</td>
<td>164</td>
<td>505</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>128</td>
<td>328</td>
<td>1,028</td>
<td></td>
</tr>
<tr>
<td>20,000</td>
<td>268</td>
<td>664</td>
<td>2,037</td>
<td></td>
</tr>
</tbody>
</table>
## Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Native hProlog</th>
<th>Native SWI-Prolog</th>
<th>Transformed hProlog</th>
<th>Transformed SWI-Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>64</td>
<td>1,965</td>
<td>164</td>
<td>505</td>
</tr>
<tr>
<td>10,000</td>
<td>128</td>
<td>3,950</td>
<td>328</td>
<td>1,028</td>
</tr>
<tr>
<td>20,000</td>
<td>268</td>
<td>8,388</td>
<td>664</td>
<td>2,037</td>
</tr>
</tbody>
</table>

**ZIP architecture**
## Shift Runtime (ms)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Native hProlog</th>
<th>Native SWI-Prolog</th>
<th>Transformed hProlog</th>
<th>Transformed SWI-Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>64</td>
<td>1,965</td>
<td>164</td>
<td>505</td>
</tr>
<tr>
<td>10,000</td>
<td>128</td>
<td>3,950</td>
<td>328</td>
<td>1,028</td>
</tr>
<tr>
<td>20,000</td>
<td>268</td>
<td>8,388</td>
<td>664</td>
<td>2,037</td>
</tr>
</tbody>
</table>

linear in delimited stack depth
**call** (Cont) in the WAM

<table>
<thead>
<tr>
<th>Depth</th>
<th>Continuation Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>248</td>
</tr>
<tr>
<td>10,000</td>
<td>492</td>
</tr>
<tr>
<td>20,000</td>
<td>992</td>
</tr>
</tbody>
</table>

hProlog
**call((Cont) in the WAM**

<table>
<thead>
<tr>
<th>Depth</th>
<th>Continuation Call</th>
<th>Meta-Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>248</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>492</td>
<td></td>
</tr>
<tr>
<td>20,000</td>
<td>992</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{call}((\text{dummy}, \text{dummy}, \ldots, \text{dummy}))
\]
**call**(Cont) in the WAM

<table>
<thead>
<tr>
<th>Depth</th>
<th>Continuation Call</th>
<th>Meta-Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>248</td>
<td>398</td>
</tr>
<tr>
<td>10,000</td>
<td>492</td>
<td>796</td>
</tr>
<tr>
<td>20,000</td>
<td>992</td>
<td>1,586</td>
</tr>
</tbody>
</table>

**call((dummy,dummy,...,dummy))**
**call(Cont) in the WAM**

<table>
<thead>
<tr>
<th>Depth</th>
<th>Continuation Call</th>
<th>Meta-Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>5,000</td>
<td>248</td>
<td>398</td>
</tr>
<tr>
<td>10,000</td>
<td>492</td>
<td>796</td>
</tr>
<tr>
<td>20,000</td>
<td>992</td>
<td>1,586</td>
</tr>
</tbody>
</table>

linear and 1.6x faster than meta-call
Summary
Summary

- simple Prolog interface for delimited continuations
- many examples of applications
- lightweight implementation in the WAM
Ongoing/Future Work

- **additional features**
  - prompts
  - hierarchies
  - failure continuation

- **new applications**
  - tabling

- **implementation improvements**
  - program analysis (e.g., abstract interpretation)
  - program specialization
Thank You!