Delimited Continuations for Prolog

T. Schrijvers, B. Demoen, B. Desouter and J. Wielemaker

UGent, KU Leuven, UGent and Universiteit Amsterdam
Motivation
Programming patterns

Minimality

Prolog = Horn clauses + simple builtins

⇒ Lack of infrastructure to capture common patterns
Programming patterns

Minimality

Prolog = Horn clauses + simple builtins

⇒ Lack of infrastructure to capture common patterns

Solutions: meta-programming & program transformations

▶ Definite clause grammars (DCGs)
▶ Extended DCGs
▶ Structured state threading
▶ ...
Issues with non-local program transformations

1. $O(n^2)$ effort
2. Fragile w.r.t. evolution
3. New feature $\rightarrow$ impact entire system

Introduce DCGs in an existing code base to pass information through several layers.
Our solution: delimited continuations

- Program-level definitions
- More light-weight
- More robust
- No pervasive changes

M. Felleisen, The theory and practice of first-class prompts, ’88
O. Danvy & A. Filinski, Abstracting control, '90
1. Definition

2. Applications

3. Implementation in the WAM
Definition
Delimited continuation primitives

\[
p \leftarrow \\
\quad q, \\
\quad \text{writeln}(c).
\]

\[
q \leftarrow \\
\quad \text{writeln}(a), \\
\quad \text{writeln}(b).
\]
Delimited continuation primitives

\[
p : \leftarrow \\
\quad q, \\
\quad \text{writeln}(c).
\]

\[
q : \leftarrow \\
\quad \text{writeln}(a), \\
\quad \text{writeln}(b).
\]

\[
? \leftarrow p. \\
\quad a \\
\quad b \\
\quad c
\]
Delimited continuation primitives

\[
\begin{align*}
\text{p} & : - \\
& \quad \text{reset}(q, _, _), \\
& \quad \text{writeln}(c). \\
\text{q} & : - \\
& \quad \text{writeln}(a), \\
& \quad \text{shift}(\_), \\
& \quad \text{writeln}(b). \quad \leftarrow \text{Continuation}
\end{align*}
\]
Delimited continuation primitives

reset /3 and shift /1

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

\[ q : \quad \text{writeln}(a), \text{shift}(_), \text{writeln}(b). \quad \leftarrow \text{Continuation} \]
Delimited continuation primitives

\[ p : \]
\[ \text{reset}(q, \_ , \text{Term}) , \]
\[ \text{writeln}(\text{Term}) , \]
\[ \text{writeln}(c) . \]

\[ q : \]
\[ \text{writeln}(a) , \]
\[ \text{shift}(\text{hello}) , \]
\[ \text{writeln}(b) . \]
Delimited continuation primitives

\[
\begin{align*}
p & : - \\
   & \quad \text{reset}(q, -, \text{Term}), \\
   & \quad \text{writeln}(	ext{Term}), \\
   & \quad \text{writeln}(c).
\end{align*}
\]

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

\[
\begin{align*}
? - p. \\
\text{a} \\
\text{hello} \\
\text{c}
\end{align*}
\]
Delimited continuation primitives

\[ p : \]
\[ \quad \text{reset}(q, \text{Cont}, \_), \]
\[ \quad \text{writeln}(c), \]
\[ \quad \text{call}(\text{Cont}). \]

\[ q : \]
\[ \quad \text{writeln}(a), \]
\[ \quad \text{shift}(\_), \]
\[ \quad \text{writeln}(b). \]
Delimited continuation primitives

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

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

?— p.
  a
  c
  b
```
Delimited continuation primitives

\[ p :\]
\[ \text{reset}(q, \text{Cont}, \_), \]
\[ \text{writeln}(c), \]
\[ \text{call}(\text{Cont}). \]

\[ q :\]
\[ \text{writeln}(a), \]
\[ \text{shift}(\_), \]
\[ \text{writeln}(b). \]
Applications
Effect handlers
Implicit state passing

\[
\text{inc} :\quad \text{get}(S), \quad S_1 \text{ is } S + 1, \quad \text{put}(S_1).
\]

?\quad \text{run\_state}((\text{inc}, \text{inc}), 0, S).
\quad S = 2.

G. Plotkin & M. Pretnar, Handlers of algebraic effects, ’09
Implicit state passing

g_{et}(S) \leftarrow \text{shift}(g(S)).

g_{ut}(S) \leftarrow \text{shift}(p(S)).
get(S) :- shift(g(S)).
put(S) :- shift(p(S)).

run_state(Goal, Sin, Sout) :-
  reset(Goal, Cont, Command),
  ( Cont = 0 ->
    Sout = Sin ;
    Command = g(S) ->
    S = Sin,
    run_state(Cont, Sin, Sout)
  ;
  Command = p(S) ->
    run_state(Cont, S, Sout)
).
Definite clause grammars

- Well-known Prolog extension
- Sequential access to elements of implicit list
- Conventionally: program transformation
Definite clause grammars

- Well-known Prolog extension
- Sequential access to elements of implicit list
- Conventionally: program transformation

$\text{Grammar (ab)}$

\[
\begin{align*}
\text{ab}(0) & . \\
\text{ab}(N) : & - c(a), c(b), \text{ab}(M), N \\
& \text{is } M + 1 . \\
\text{phrase(ab(N), [a, b, a, b], [])} & . \\
N = 2 .
\end{align*}
\]
Definite clause grammars

- Well-known Prolog extension
- Sequential access to elements of implicit list
- Conventionally: program transformation

Grammar \((ab)^n\)

\[
ab(0).
\]

\[
ab(N) \leftarrow \text{c(a)}, \text{c(b)}, \text{ab}(M), N \text{ is } M + 1.
\]

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

\[
N = 2.
\]
Definite clause grammars

c(E) :- shift (c(E)).

phrase (Goal, Lin, Lout) :-
    reset (Goal, Cont, Term),
    ( Cont == 0 ->
      Lin = Lout
    ; Term = c(E) ->
      Lin = [E|Lmid],
      phrase (Cont, Lmid, Lout)
    ).
Applications

- Implicit state
- Implicit environments
- Logging
- Exceptions: alternate semantics
- Iterators
- Iteratees
- Coroutines
- Search heuristics (Tor)
- ...
Composing effect handlers

- Effect handlers easily made compositional
- Propagate unknown operations

Counting ab

\[ ab \. \]
\[ ab :\! \! - c(a) , c(b) , inc , ab. \]
\texttt{phrase} \hspace{0.5cm} (\texttt{Goal}, \texttt{Lin}, \texttt{Lout}) :\texttt{-}
\texttt{reset} \hspace{0.5cm} (\texttt{Goal}, \texttt{Cont}, \texttt{Term}),
\begin{align*}
( \texttt{Cont} &\equiv 0 \rightarrow \ldots \\
\texttt{Term} = \texttt{c} (\texttt{E}) &\rightarrow \ldots \\
\texttt{shift} \hspace{0.5cm} (\texttt{Term}),
\texttt{phrase} \hspace{0.5cm} (\texttt{Cont}, \texttt{Lin}, \texttt{Lout})
\end{align*}
) \hspace{0.5cm} .
Composing effect handlers

run :-
    run_state(
        phrase(ab,[a,b,a,b],[]),
        0,
        N),
    write(N).

?- run.
2
Implementation
Catch&Throw versus Reset&Shift

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

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

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

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

Goal :— shift(Term).
?— reset(Goal, Cont, Ball), . . .

▶ unify Term with Ball
▶ leave the stacks intact (protect them)
▶ unify Cont with copy of the environments up to reset /3
▶ control goes to . . .
Four Issues

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

- catch-throw does this
- so, re-use same principle
- a marker in code or a marker in the environment
- follow the chain of environments, starting at $E$
catch-throw does this

so, re-use same principle

a marker in code or a marker in the environment

follow the chain of environments, starting at $E$

protect the stacks: fiddle with the TOS pointer

easy peasy for systems with stack freezing
Copying a delimited part of the environment stack

**caveat:** it’s not a deep copy!

- if continuation is first-class citizen: easy - e.g. BinProlog

- in WAM more complicated:
  - env contains **code pointer, env pointer** and terms
  - copying whole environment as is can be arbitrarily bad in terms of live environment slots
  - so, want selective copy

- determine live env slots
  - live variable map (hProlog, YAP, . . .)
  - decompilation based (SWI-Prolog, SICStus Prolog, . . .)
Copying a delimited part of the environment stack II

The copied environment is a normal Prolog term:

\[$cont$(abstraction_of(CP), 2-[Y] , 2)$]
Copying a delimited part of the environment stack II

a(X) :- b, c(X,Y), shift(1), d(Y).

The copied environment is a normal Prolog term:

$cont$(abstraction_of(CP), 2-[Y], 2)

A copy of a continuation chain = a list of $cont$/2 terms
Using a copied continuation

Built-in call_tailbody/1 uses $\text{cont}$(abstraction_of(CP), [Y])

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

\[ \text{Env}(a/1) \]

\[ \begin{array}{c}
???
\\
???
\\
Y
\end{array} \]

P
Fineprint

- abstraction_of(CP)
- non-determinism
- cut
- executing a continuation twice
- errors
- interaction with catch&throw ...
Performance

- silly :-)
- catch&throw: linear in depth
- reset & shift : linear in depth and number of variables
Performance

- silly :-)
- catch&throw: linear in depth
- reset & shift: linear in depth and number of variables

<table>
<thead>
<tr>
<th>goals</th>
<th>hProlog</th>
<th>SWI-Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>catch(throw(1),<em>,true)/reset(shift(1),</em>,_)</td>
<td>2.5/1</td>
<td>2/1</td>
</tr>
<tr>
<td>depth 10 - no variables</td>
<td>1.4/1</td>
<td>1/3</td>
</tr>
<tr>
<td>depth 10 - n (live) variables - break even</td>
<td>n = 4.5</td>
<td>–</td>
</tr>
</tbody>
</table>
Summary

Many **useful applications**: implicit state, implicit environments, logging, nonbacktracking exceptions, iterators, iteratees, coroutines, search heuristics, . . .

**Program-level definitions**

**Lightweight implementation in the WAM**

- mostly independent
- performance OK
Do you have . . .

Questions?
Do you have . . .

Questions?

Answers?

Suggestions?