Delimited Continuations for Prolog

Tom Schrijvers

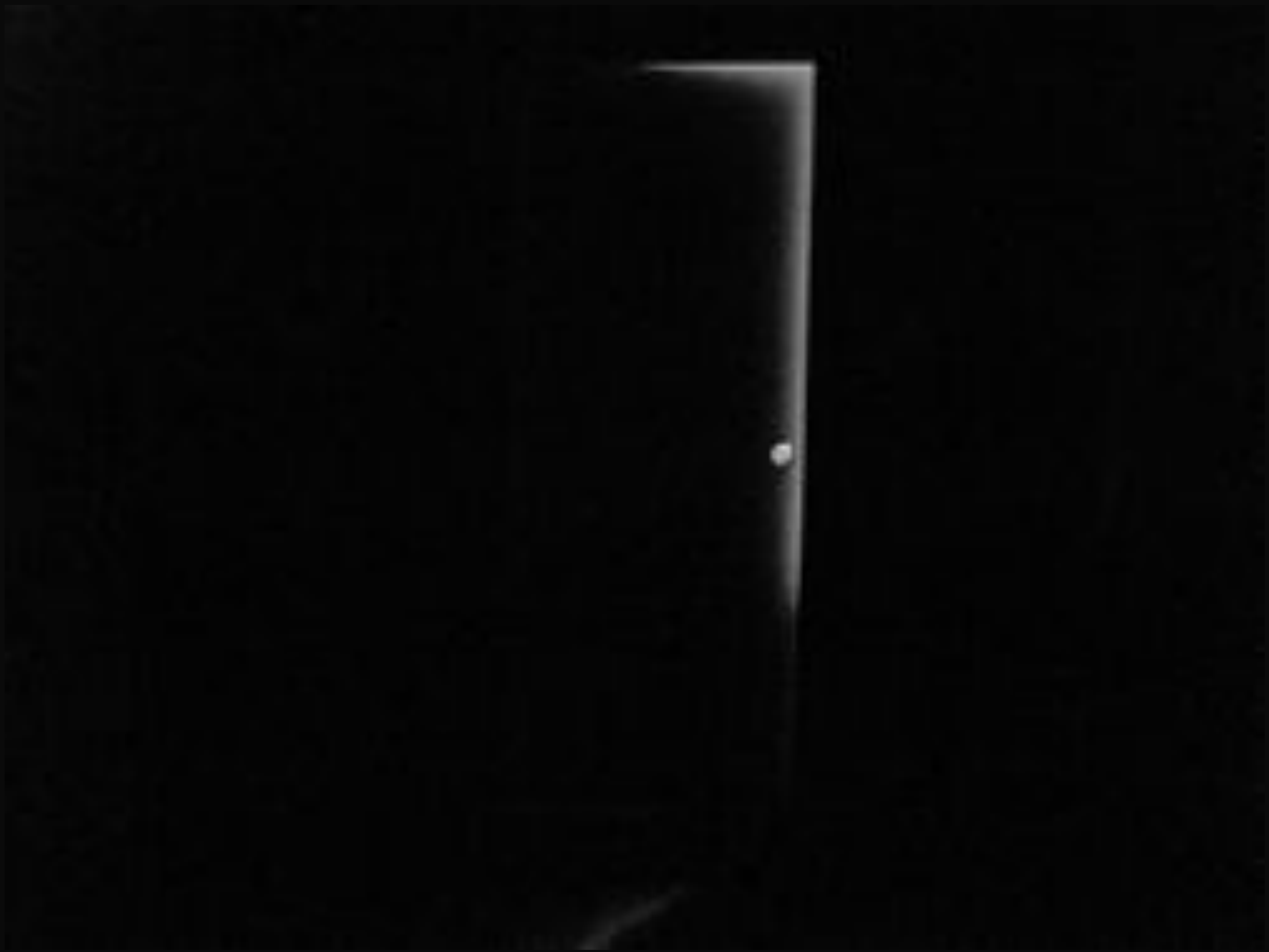
UNIVERSITEIT GENT

Kiel Declarative Programming Days 2013

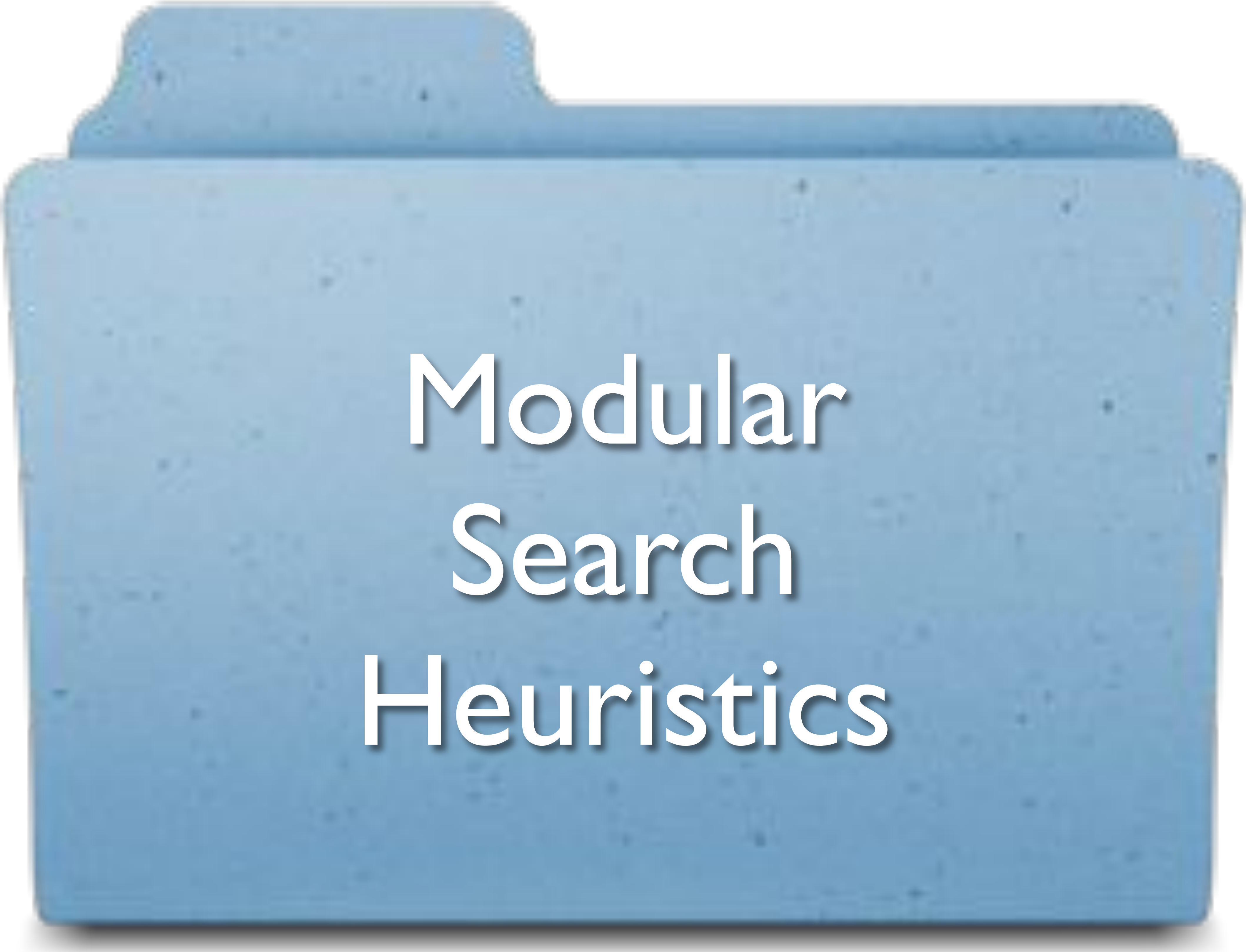
Motivation

Delimited Continuations

- from Functional Programming
  - Felleisen POPL'88
  - Danvy & Filinski LFP'90
- greatly underused and underappreciated



Prolog lacks infrastructure to capture control patterns



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

Under consideration for publication in Theory and Practice of Logic Programming 1

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

ICLP 2013

Benoit Desouter

Bart Demoen

Jan Wielemaker

Many Uses

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

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(Goal,Continuation,Term)

shift(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.
```

Plain Reset

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

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

```
?- main.
```

Plain Reset

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

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

```
?- main.
```

Plain Reset

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

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

```
?- main.
a
```

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

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

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

```
?- main.
```

Aborting

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

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

```
?- main.
```

Aborting

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

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

```
?- main.
```

Aborting

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

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

```
?- main.
a
```

Aborting

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

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

```
?- main.
a
```

Aborting

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

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

```
?- main.
a
c
```

add transitions

Term Passing

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

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

```
?- main.
```

add transitions

Term Passing

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

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

```
?- main.
a
```

add transitions

Term Passing

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

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

```
?- main.
a
hello
```

add transitions

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
```

add transitions

Repeated Call

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

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

```
?- main.
```

add transitions

Repeated Call

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

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

```
?- main.
a
```

add transitions

Repeated Call

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

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

```
?- main.
a
c
```

add transitions

Repeated Call

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

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

```
?- main.
a
c
b
```

add transitions

Repeated Call

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

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

```
?- main.
a
c
b
b
```

No Shift

```
?- reset(true,Cont,Term).
```

No Shift

```
?- reset(true,Cont,Term).
Cont = 0,
Term = 0.
```

No Reset

```
?- shift(x).
```

No Reset

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

add transitions

Backtracking

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

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

```
?- main.
```

add transitions

Backtracking

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

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

```
?- main.
c
```

add transitions

Backtracking

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

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

```
?- main.
c
a
```

add transitions

Backtracking

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

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

```
?- main.
c
a ;
```

add transitions

Backtracking

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

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

```
?- main.
c
a ;
c
```

add transitions

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

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

static program transformation

```
ab(L,L).
ab([a,b|L],T) :- ab (L,T).
```

Program Transformation

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

static program transformation

```
ab(L,L).
ab([a,b|L],T) :- ab (L,T).
```

```
phrase(G,L,T) :- call(G,L,T).
```

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

command

```
c/1                        phrase/3

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

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

DCGs

command

handler

```
c/1                          phrase/3

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

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

DCGs

command

handler

```
c/1
```

```
phrase/3
```

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

example code

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

DCGs

command

handler

```
c/1
```

```
phrase/3
```

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

example code

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

example query

Syntax

```
c(X) :- shift(c(X)).
```

Semantics: Handler

```
phrase(G,L,T) :-
  reset(G,Cont,Command),
  ( Command = c(X) ->
        L = [X|NL],
        phrase(Cont,NL,T)
  ;
        L = T
  ).
```


Implicit
State

Implicit State

```
get/1, put/1          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)).
```

Handler

```
runState(G,Sin,Sout) :-
  reset(G,Cont,Command),
  ( Command = get(S) ->
      S = Sin,
      runState(Cont,Sin,Sout)
  ; Command = 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.
```

Alternative Handler

```
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

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

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

```
?- runState(
       phrase(ab,[a,b,a,b],[]),
       0,S).
S = 2.
```

Example

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

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

```
?- phrase(
        runState(ab,0,S),
        [a,b,a,b],[]).
S = 2.
```

Compositional Handler

```
phrase(G,L,T) :-
  reset(G,Cont,Command),
  ( Command = c(X) ->
      L = [X|NL],
      phrase(Cont,NL,T)
  ; Command = 0 ->
      L = T
  ;
      shift(Command),
      phrase(Cont,L,T)
  ).
```

Many Uses

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

This Talk

Semantics

Applications

Implementation

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

```
eval(+Goal,-Status)

Status:
- ok
- shift(Term,Cont)
```

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

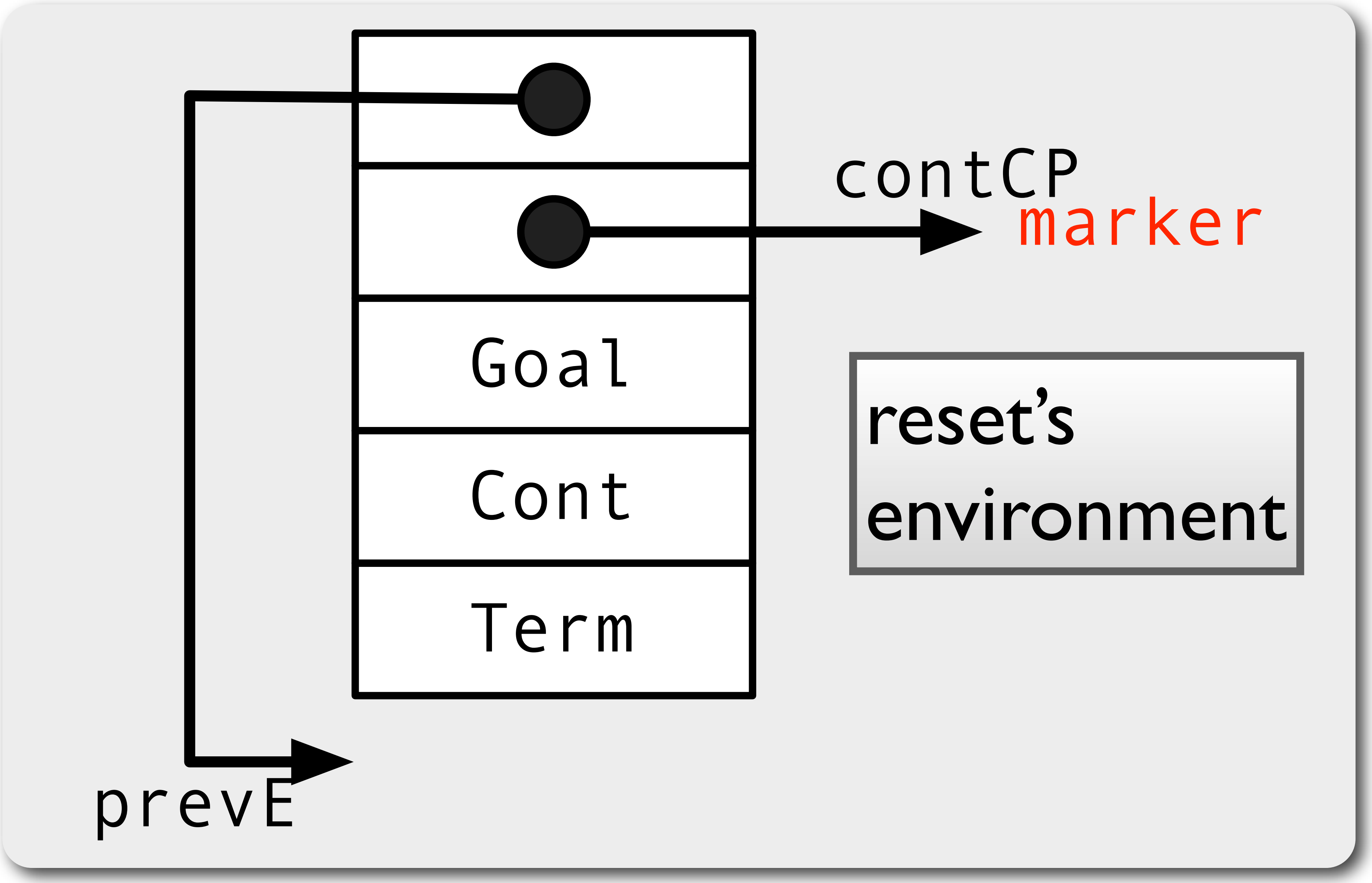
```
Goal :- ... shift(Term) ...
?- reset(Goal,Cont,Ball), ...
```

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

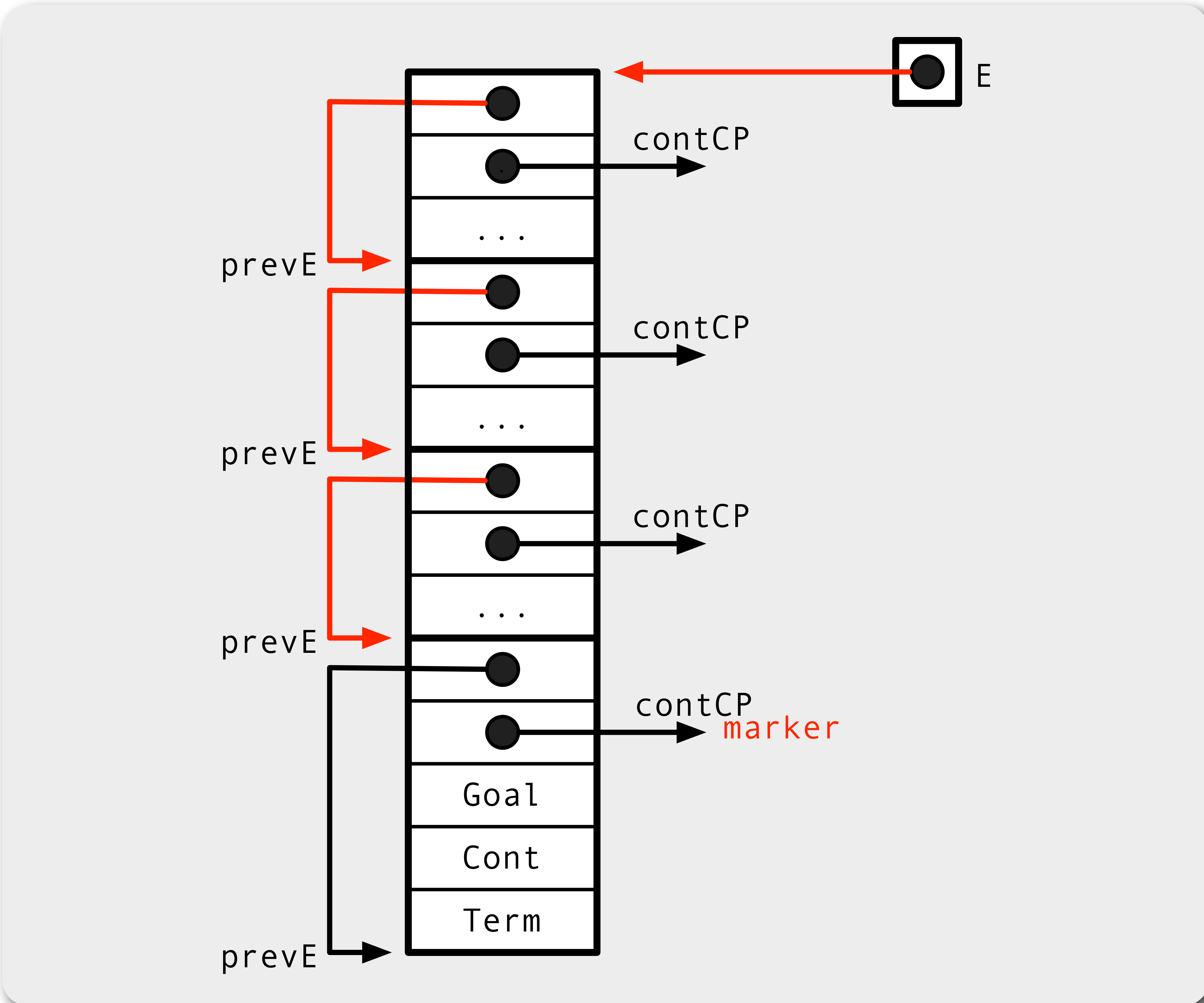
1. up to `reset/3`
2. how to copy (a delimited part of) the environment stack
3. how to use this delimited continuation
4. fineprint

Up to reset/3



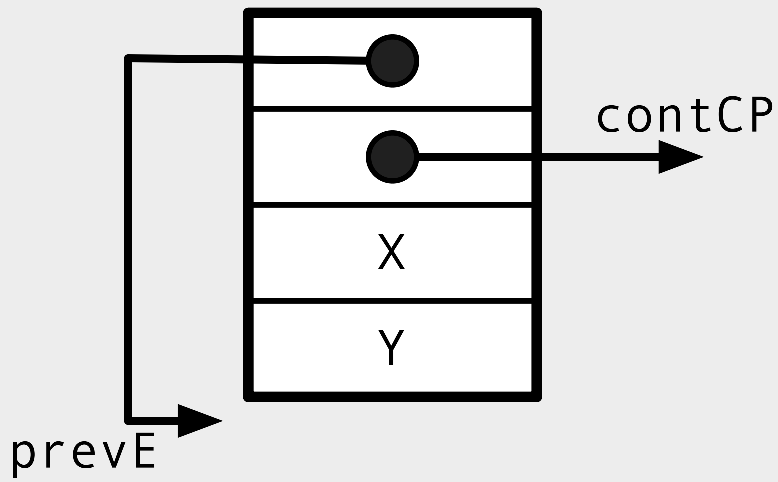
same principle as catch/throw

Up to reset/3


E
contCP
...
prevE
contCP
...
prevE
contCP
...
prevE
contCP
marker
Goal
Cont
Term
prevE

Continuation Term

Environment Stack



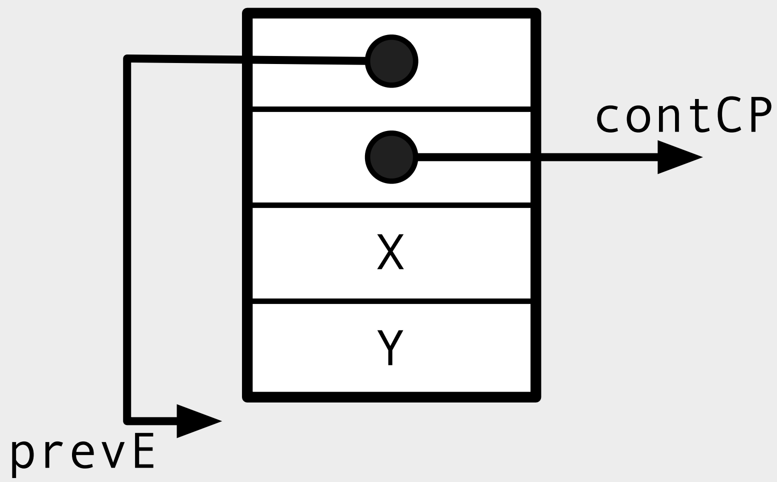
Heap

```
$cont$(ContCP,[X,Y])
```

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

Reified Environment

Environment Stack

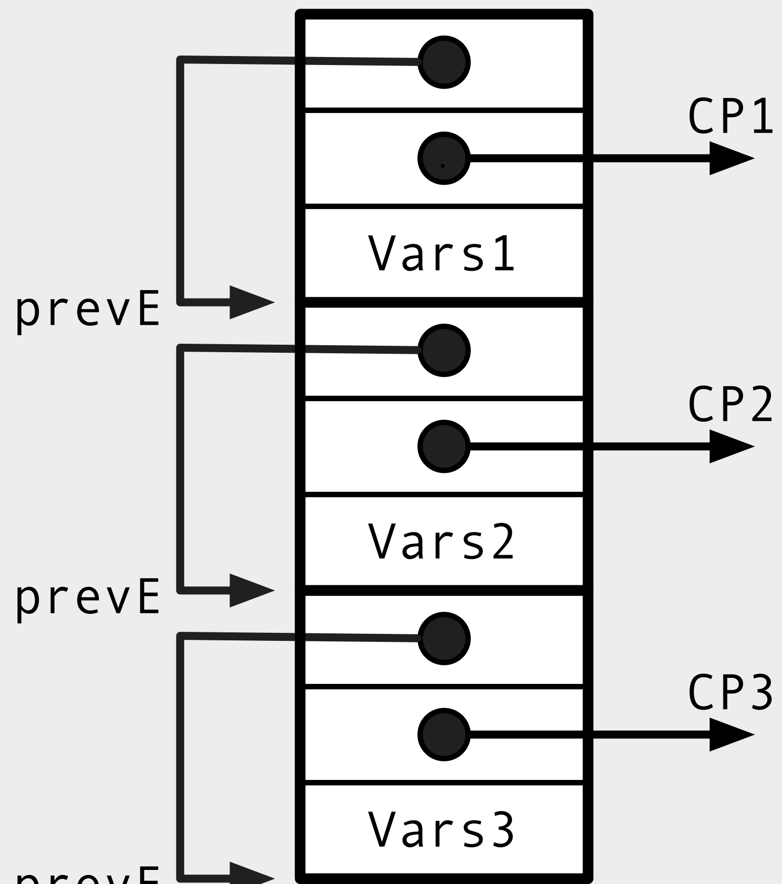


Heap

```
$cont$(ContCP,[X,Y])
```

Environment Chain

Environment Stack



Heap

```
[$cont$(CP1,Vars1),
 $cont$(CP2,Vars2),
 $cont$(CP3,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)

Depth
5,000
10,000
20,000

Shift Runtime (ms)

specialization of meta-interpreter

Depth	Transformed
5,000	
10,000	
20,000	

Shift Runtime (ms)

specialization of meta-interpreter

	Transformed
Depth	**hProlog**
5,000	
10,000	
20,000	

Shift Runtime (ms)

	Transformed
Depth	**hProlog**
5,000	164
10,000	328
20,000	664

Shift Runtime (ms)

	Native	Transformed
Depth		**hProlog**
5,000		164
10,000		328
20,000		664

Shift Runtime (ms)

	Native	**Transformed**
Depth	**hProlog**	**hProlog**
5,000		164
10,000		328
20,000		664

WAM architecture

Shift Runtime (ms)

	Native	Transformed
Depth	hProlog	hProlog
5,000	64	164
10,000	128	328
20,000	268	664

WAM architecture

Shift Runtime (ms)

	Native		Transformed	
Depth	**hProlog**		**hProlog**	**SWI-Prolog**
5,000	64		164	
10,000	128		328	
20,000	268		664	

Shift Runtime (ms)

	Native		Transformed	
Depth	**hProlog**		**hProlog**	**SWI-Prolog**
5,000	64		164	505
10,000	128		328	1,028
20,000	268		664	2,037

Shift Runtime (ms)

Depth	Native		Transformed	
	hProlog	SWI-Prolog		SWI-Prolog
5,000	64		164	505
10,000	128		328	1,028
20,000	268		664	2,037

ZIP architecture

Shift Runtime (ms)

	Native		Transformed	
Depth	**hProlog**	**SWI-Prolog**	ZIP architecture	**SWI-Prolog**
5,000	64	1,965	164	505
10,000	128	3,950	328	1,028
20,000	268	8,388	664	2,037

Shift Runtime (ms)

	Native		Transformed	
Depth	**hProlog**	**SWI-Prolog**	**hProlog**	**SWI-Prolog**
5,000	64	1,965	164	505
10,000	128	3,950	328	1,028
20,000	268	8,388	664	2,037

linear in delimited stack depth

`call(Cont)` in the WAM

	hProlog
Depth	**Continuation Call**
5,000	248
10,000	492
20,000	992

`call(Cont)` in the WAM

`call((dummy,dummy,...,dummy))`

Depth	Continuation Call	Meta-Call
5,000	248	
10,000	492	
20,000	992	

`call(Cont)` in the WAM

`call((dummy,dummy,...,dummy))`

Depth	Continuation Call	Meta-Call
5,000	248	398
10,000	492	796
20,000	992	1,586

`call(Cont)` in the WAM

Depth	hProlog	
	Continuation Call	Meta-Call
5,000	248	398
10,000	492	796
20,000	992	1,586

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!