Dynamic Wind for OCaml Effect Handlers with Escaping Continuation Support

SLE 2026

Antonino Yann William Gillard[1] Tetsuro Yamazaki[2]
Tomoharu Ugawa[2]

[1]IMT Atlantique

[2]University of Tokyo

July 3rd, 2026

Background

Uses of effect handlers and dynamic wind :

- coroutines, mutable states, eDSL, ...
- dynamic scope variable

Dynamic Wind for Effect Handlers (Voigt et. al 2025):

- implemented dynamic wind in Effekt
- no support for escaping continuations

Goal of this work

Implement dynamic wind as a library in OCaml on top of effect handlers while supporting escaping continuations

Effect Handlers and Dynamic Wind

Effect Handlers

- resumable exceptions
- continuation k : rest of the computation

```
try f () with
| effect (eff v), k -> continue k (v + 1)
```

Dynamic Wind

- provides safeguards when executing a computation
- preparation hook on entry,
- finalization hook on exit,
- supports entry and exit through effects

```
let dynamic_wind before main after = ...
```

Generators using escaping continuations

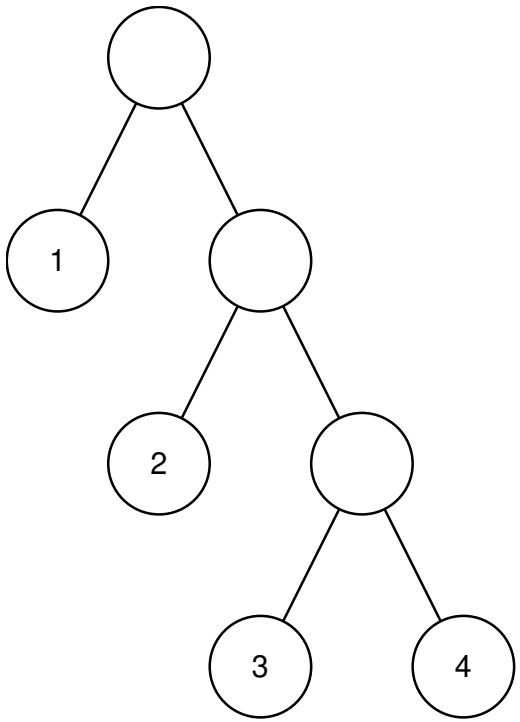
```
type _ Effect.t += Yield : int -> unit Effect.t

let gen f x : iterator =
match f x with
| v -> Done
| effect (Yield v), k -> Cont (v, continue k)
```

- `f` raises an effect `Yield`
- **gen** returns the value of `Yield` and the continuation `k`
- `k` escapes its handler's scope

Application to `same_fringe`

Compares that 2 trees `t1` and `t2` have the same fringe.



Tree of fringe [1, 2, 3, 4]

`visit` traverses the tree and yields the leaf values

```
let same_fringe t1 t2 =
  let it1 = gen visit t1 in
  let it2 = gen visit t2 in
  match it1, it2 with
  | Done, Done -> True
  | Cont(v1, k1), Cont(v2, k2)
    -> ...
  | _, _ -> False
```

Logging mechanism

We add a logger function to study the behaviour of `visit`.

```
let current_logger = ref None
```

`visit` now calls `current_logger` on each leaf value. We only need to study one tree, so we add the option to activate / deactivate the logger.

```
let with_logger f logger =
  let old_logger = ref None in
    dynamic_wind
      (fun () ->
      old_logger := !current_logger;
      current_logger := logger)
      f
      (fun () -> current_logger := !old_logger)
```

same_fringe with logger

```
let same_fringe t1 t2 =
  let it1 = with_logger
    (fun () -> gen visit t1) logger
  in
  let it2 = gen visit t2 in
  match it1, it2 with
  | Done, Done -> True
  | Cont(v1, k1), Cont(v2, k2)
    -> ...
  | _, _ -> False
```

Intuition : `logger` active only when traversing `t1`

However, `logger` is only active for `t1`'s first leaf !

Problem

Dynamic wind context :
Sequence of dynamic wind frames whose preparation hooks have been executed but corresponding finalization hooks have not

Problem

The dynamic wind context is not preserved across suspension and resumption of computation.

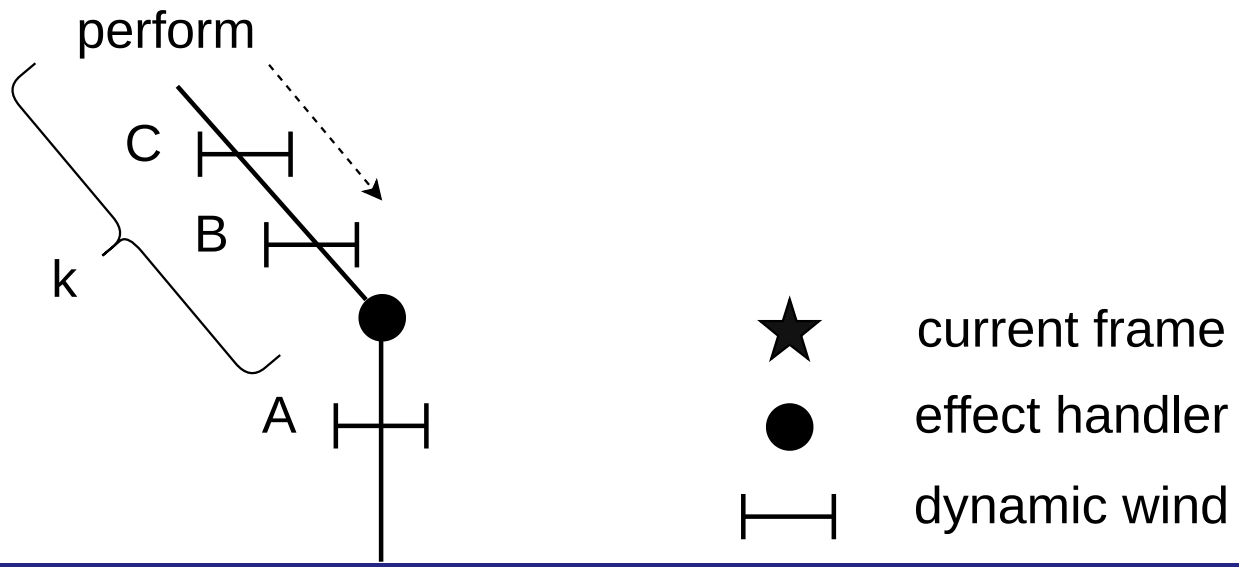
Goal

Adapt dynamic wind to support escaping continuations.
Continuations captured in dynamic wind context should resume in the same scope

Delimited continuation semantics

Dynamic wind behaves with respect to the control stack constructed by delimited continuations

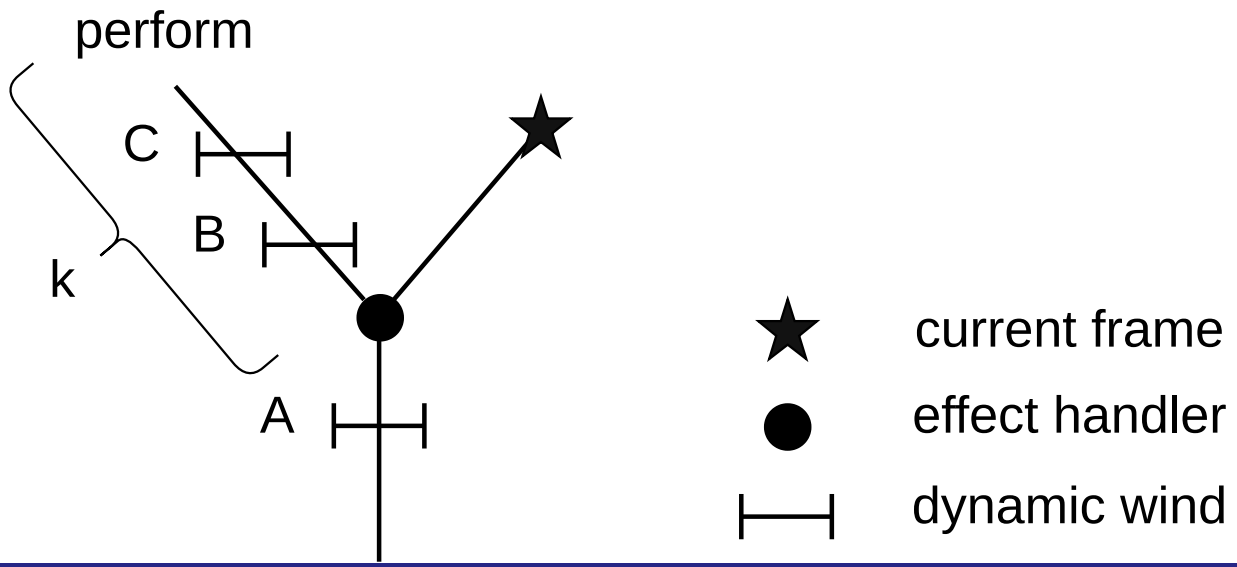
```
try ... (* dynamic wind B and C *)
  perform Effect
```



Delimited continuation semantics

Dynamic wind behaves with respect to the control stack constructed by delimited continuations

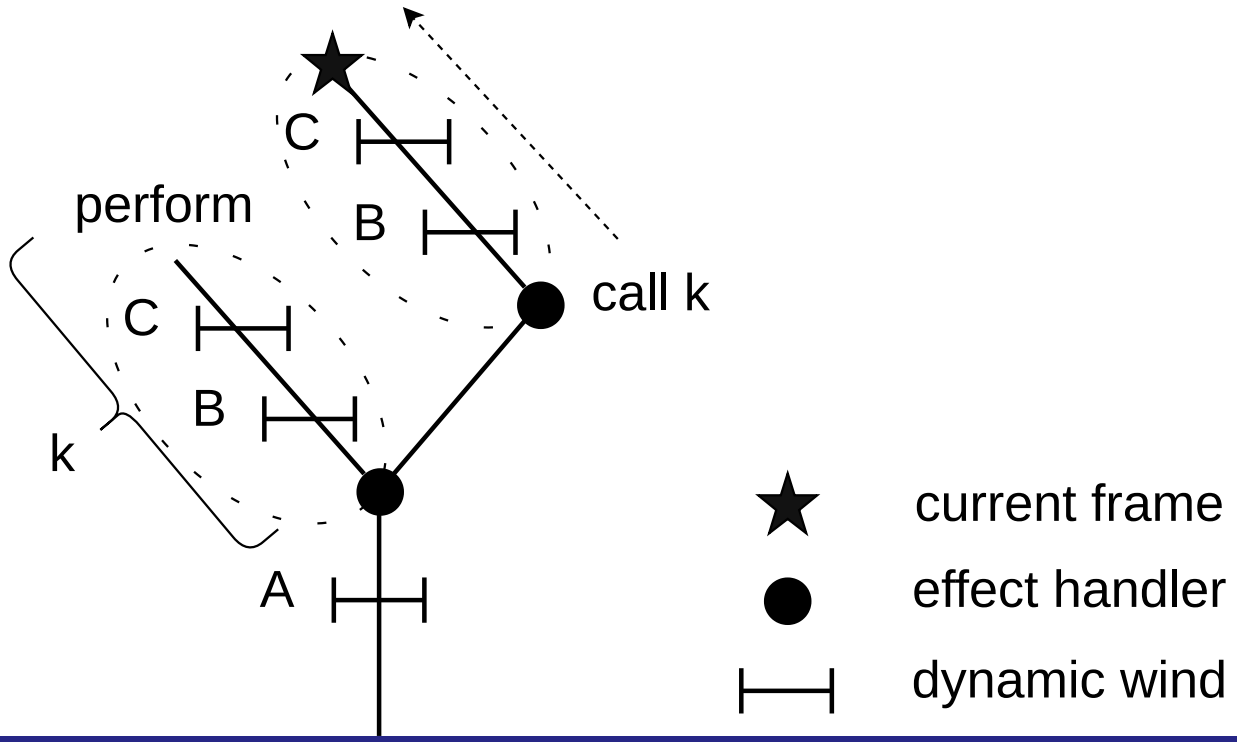
```
try ... with
| effect Effect, k -> ...
```



Delimited continuation semantics

Dynamic wind behaves with respect to the control stack constructed by delimited continuations

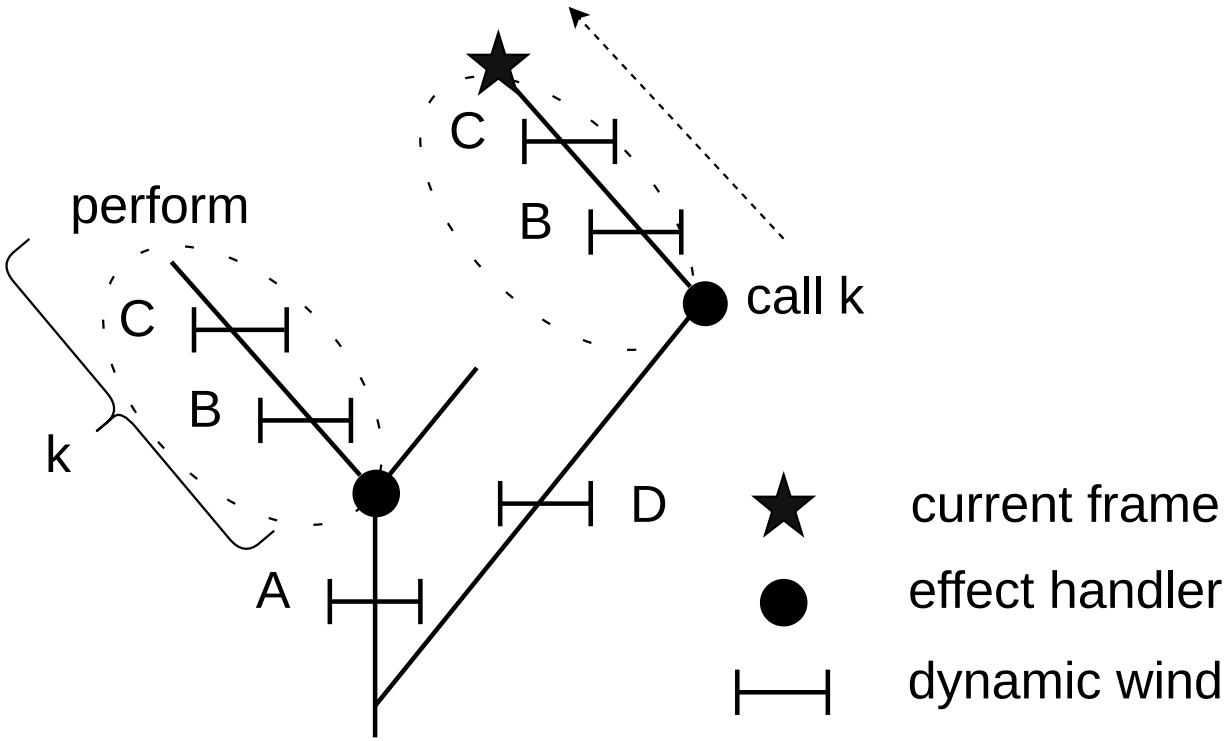
```
try ... with
| effect Effect, k -> ... continue k ()
```



Delimited continuation semantics

Dynamic wind behaves with respect to the control stack constructed by delimited continuations

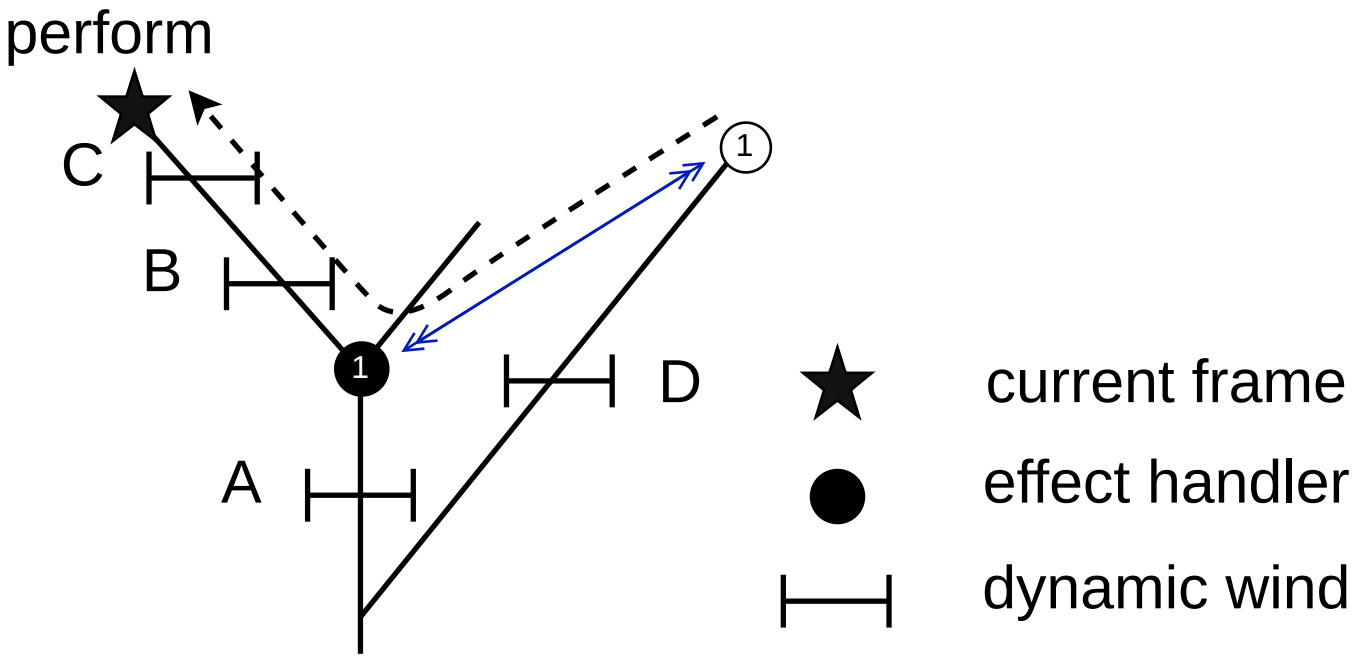
```
let f = try ... with
| effect Effect, k -> ... continue k
in dynamic_wind before (fun () -> f ()) after
```



Full continuation semantics

Hooks are executed so that the dynamic wind context is **preserved**

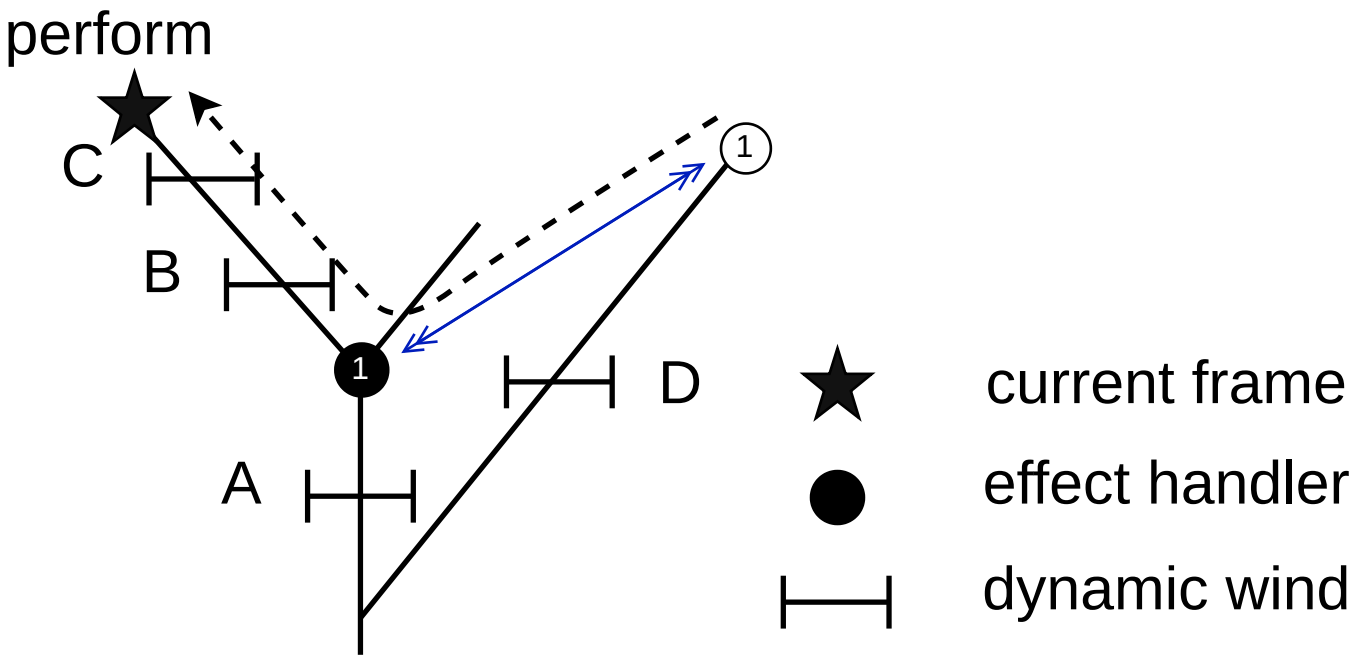
Wormhole links (⇜⇝): Control transfers between effect handlers and where its continuation resumes



Control-Transfer Cactus Stack Model

Full continuation semantics

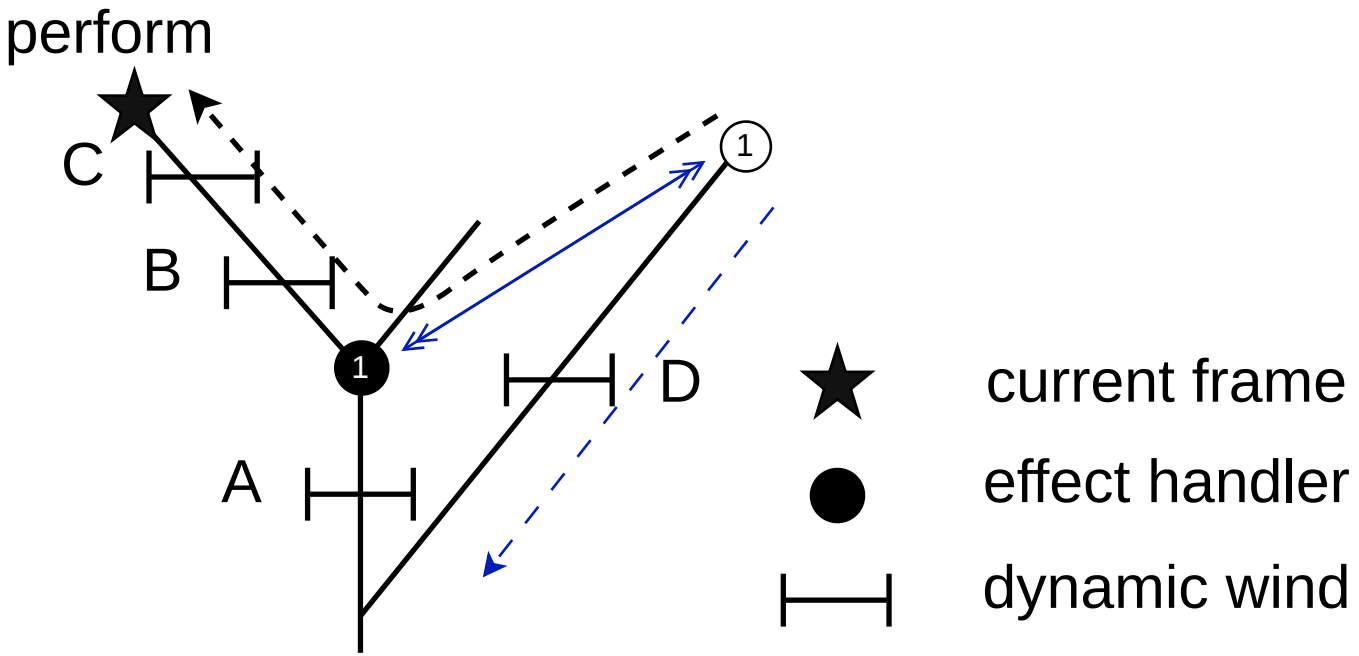
Dynamic wind track : Walk on the cactus stack obtained by replacing each ⇜⇝link in the return path with their detours



Control-Transfer Cactus Stack Model

Full continuation semantics

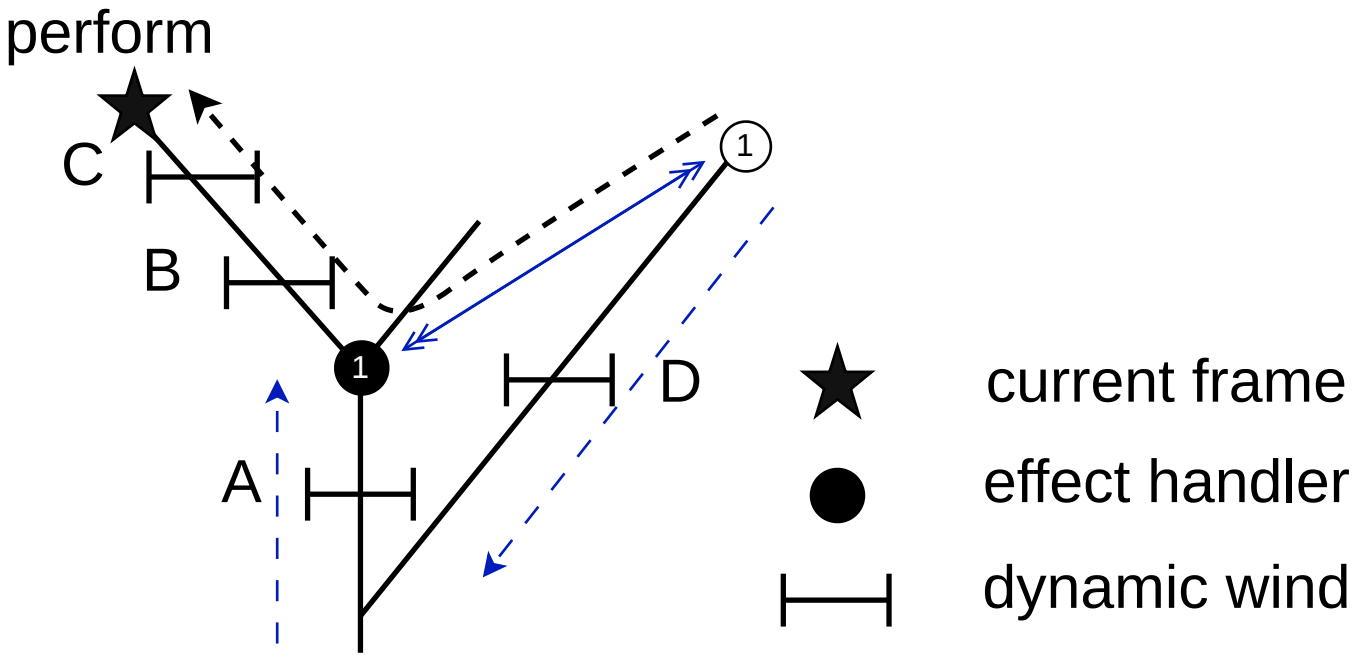
Dynamic wind track : Walk on the cactus stack obtained by replacing each ⇜⇝link in the return path with their detours



Dynamic wind track of ⇜⇝(1/2)

Full continuation semantics

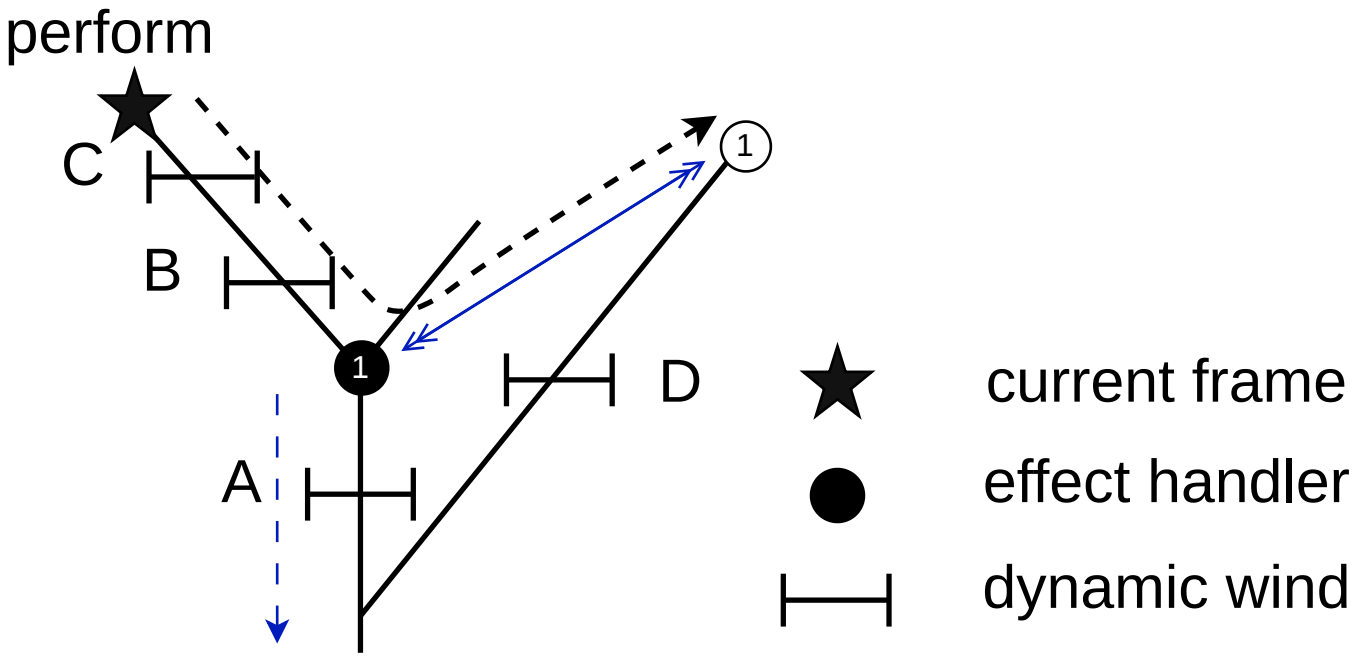
Dynamic wind track : Walk on the cactus stack obtained by replacing each ⇜⇝link in the return path with their detours



Dynamic wind track of ⇜⇝(2/2)

Full continuation semantics

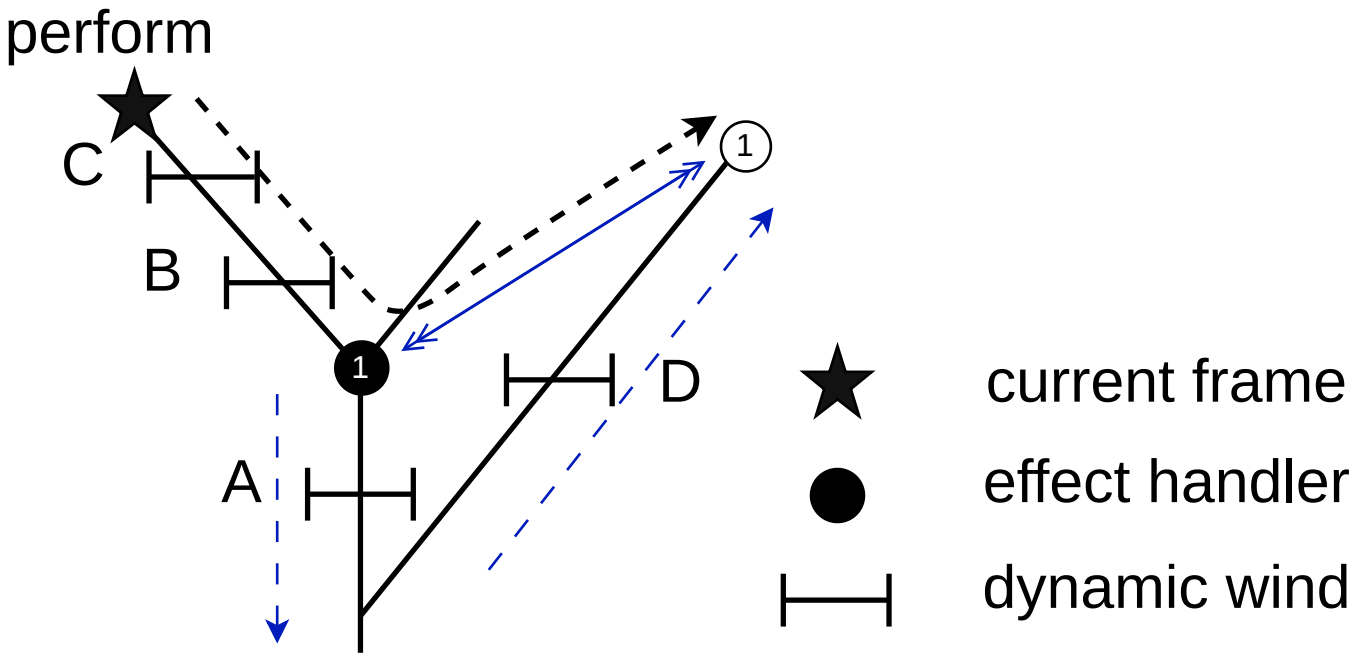
Dynamic wind track : Walk on the cactus stack obtained by replacing each ⇜⇝link in the return path with their detours



Return path (1/3)

Full continuation semantics

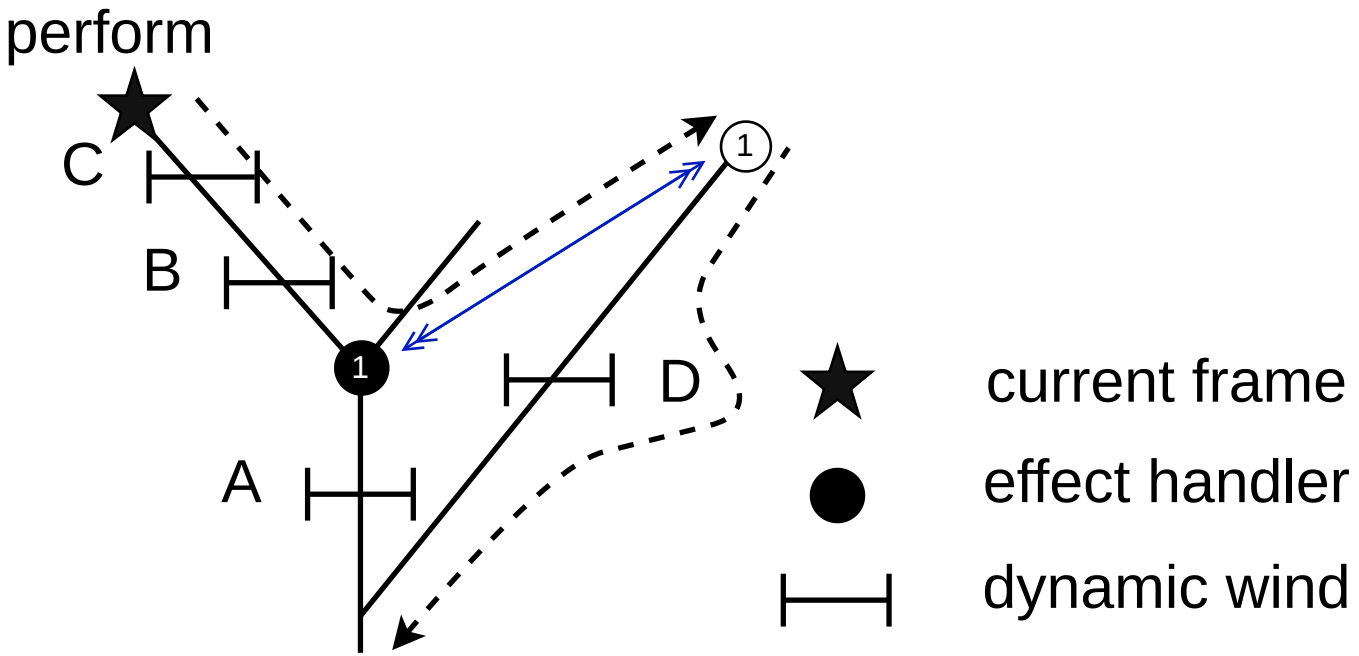
Dynamic wind track : Walk on the cactus stack obtained by replacing each ⇜⇝link in the return path with their detours



Return path (2/3)

Full continuation semantics

Dynamic wind track : Walk on the cactus stack obtained by replacing each ⇜⇝link in the return path with their detours

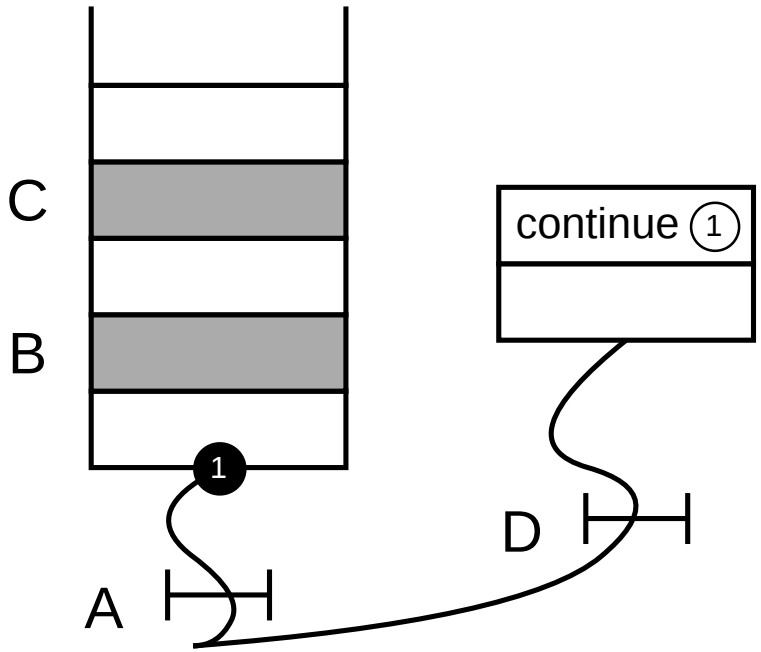


Return path (3/3)

Implementation in OCaml

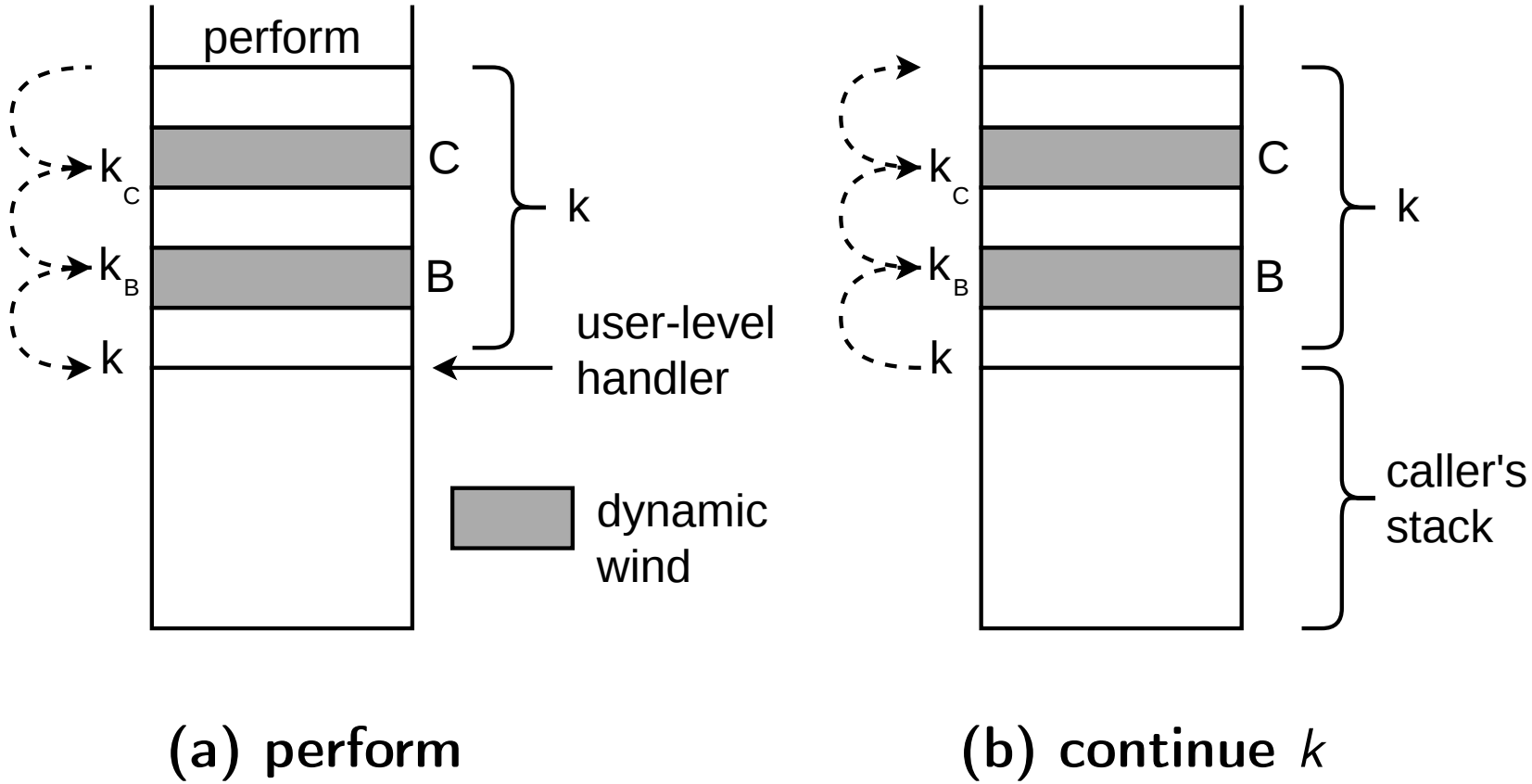
Keeping track of dynamic wind context :

- Within a continuation
- When going through a $\twoheadleftarrow\twoheadrightarrow$link



Within a continuation

Dynamic wind uses an effect handler which catches **any** effect.
Each effect caught is then **performed again**.



(a) perform **(b) continue *k***

Wormhole link

`dw_stack` : stack of dynamic wind hooks updated on execution of each hook

Decorating continuations :

```
| effect e, k -> wrap (fun k -> ...) k
```

`wrap` captures the dynamic wind context at the effect handler.

Resuming the decorated continuation :

- `continue` captures the dynamic wind context on resumption
- dynamic wind track : execute the **stack difference** between `wrap` and `continue`

Optimize depending on dynamic wind context.

Evaluation

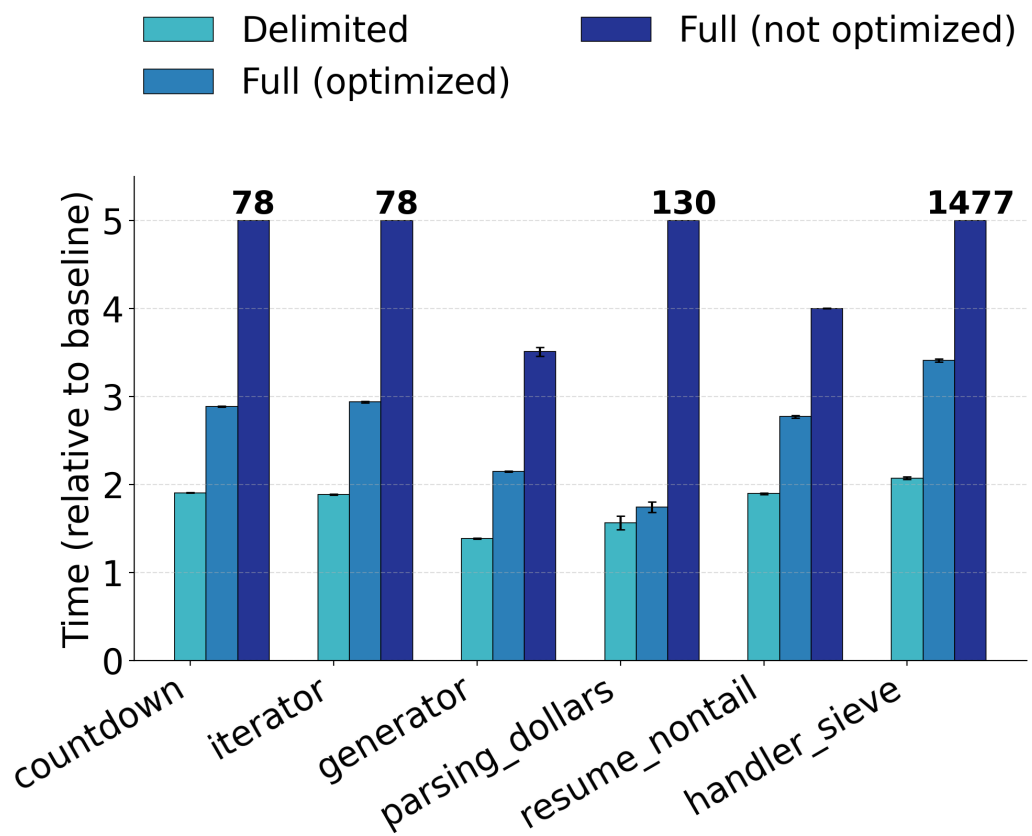
Goal : Measuring the overhead induced by full continuation semantics

- Measure it in its intended clauses
- Measure it in other clauses

Method : Measuring the overhead induced by full continuation semantics

- Preparation hook : `fun () -> ()`
- Finalization hook : `fun () -> ()`

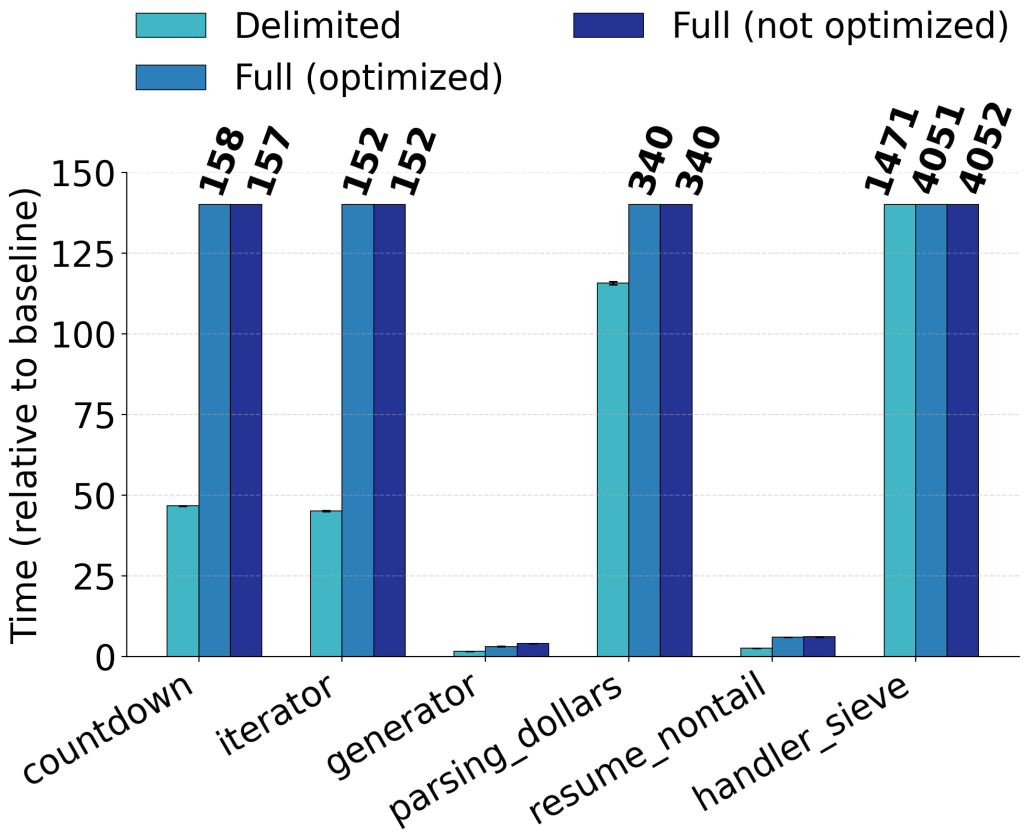
Results : Intended cases



Dynamic wind in `try` clauses

Optimization : 19x as fast on the geometric mean

Results : Unintented cases



Dynamic wind in `handler` clauses

Reason of overhead : Breaking tail call optimizations

Related Works

Constraining Control (Friedman & Haynes, 1985):

- Original notion of dynamic wind in Scheme

Adding delimited and composable control to a production programming environment (Flatt et al. 2007):

- Dynamic wind for multi-shot continuations under delimited continuation semantics

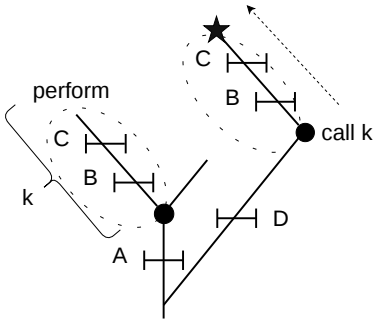
Dynamic Wind for Effect Handlers (Voigt et al. 2025):

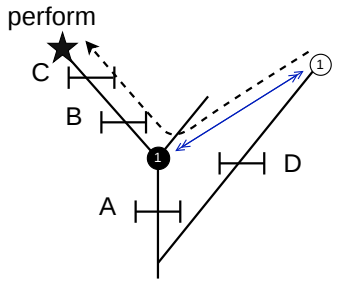
- Dynamic wind for the Effekt language under delimited continuation semantics

Conclusion

Contributions:

- Identified unintuitive interactions between dynamic wind and escaping continuations
- New semantics decorating single-shot continuations with a list of enclosing dynamic winds
- Acceptable overhead in `try` clauses





Future work:

- Reduce overhead in effect clauses
- Formal semantics
- Multi-shot continuation support