Shift to control

Chung-chieh Shan

"The continuation represents an entire (default) future for the computation."

The delimited continuation represents a partial (default) future for the computation.

"The continuation represents an entire (default) future for the computation."

The delimited continuation represents a partial (default) future for the computation.

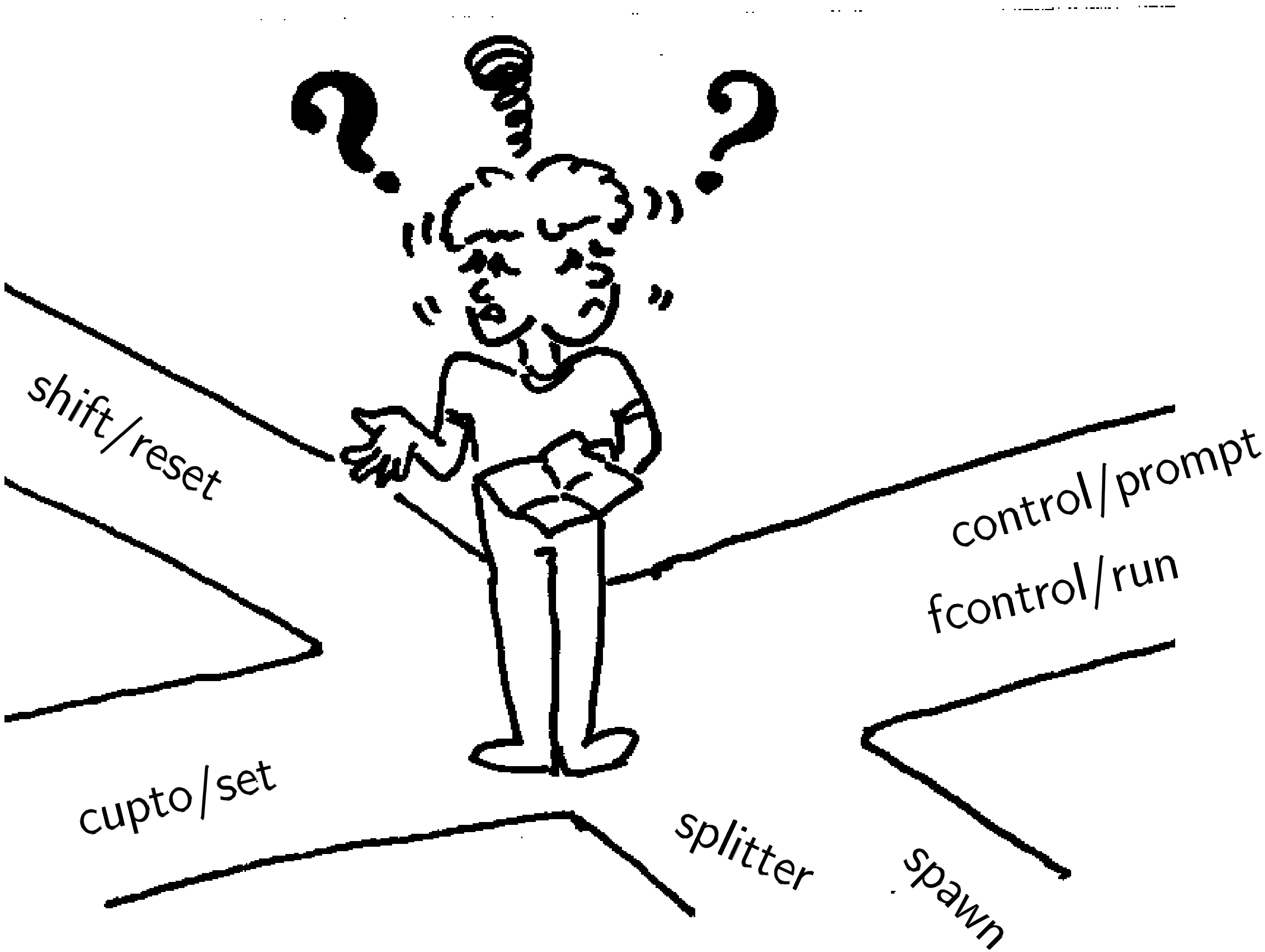
Some applications of undelimited and delimited continuations:

- generators
- representing monads
- partial evaluation
- Web interactions
- mobile code
- the CPS transform itself
- linguistics
- ...

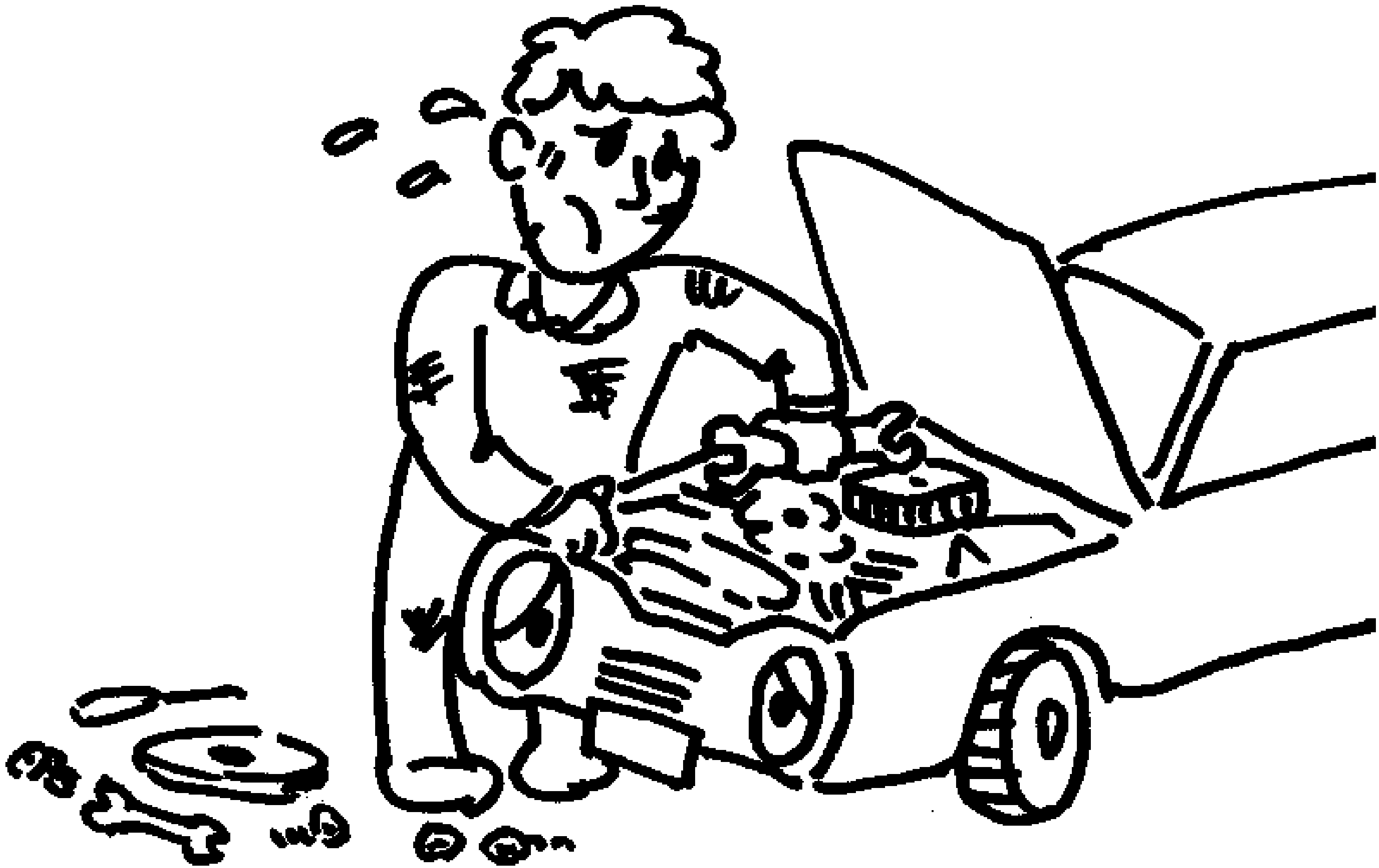
Not to mention the good old days of same-fringe!

"Control operators make my head hurt!"

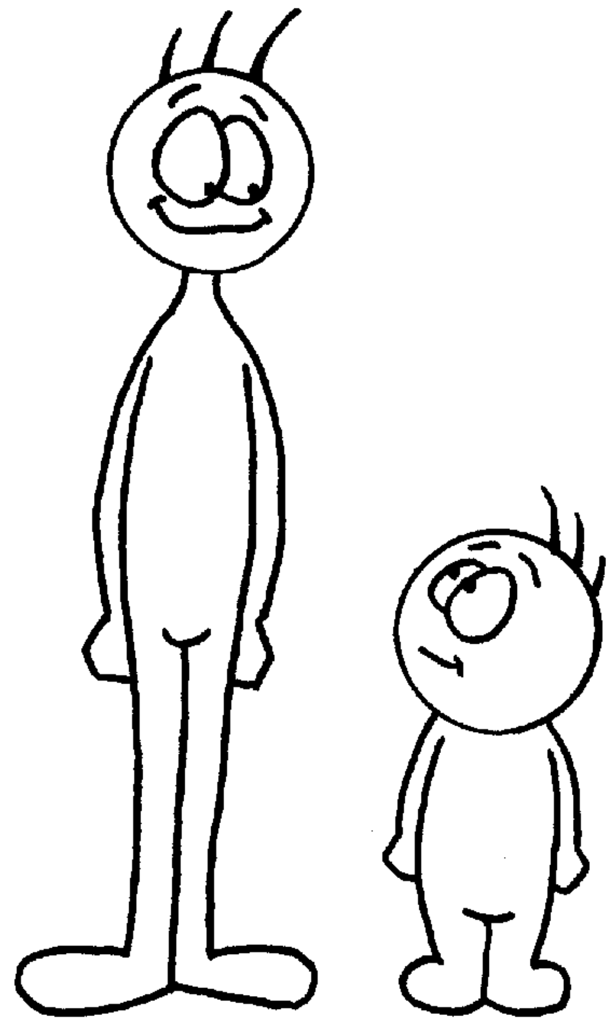
There are so many delimited control operators to choose from,
it's hard to know which ones to use.



Or which ones to implement!



It is also unclear how some operators relate to continuation-passing style.



This paper answers two related questions.

This paper answers two related questions.

Gasbichler and Sperber implement shift/reset and control/prompt directly.

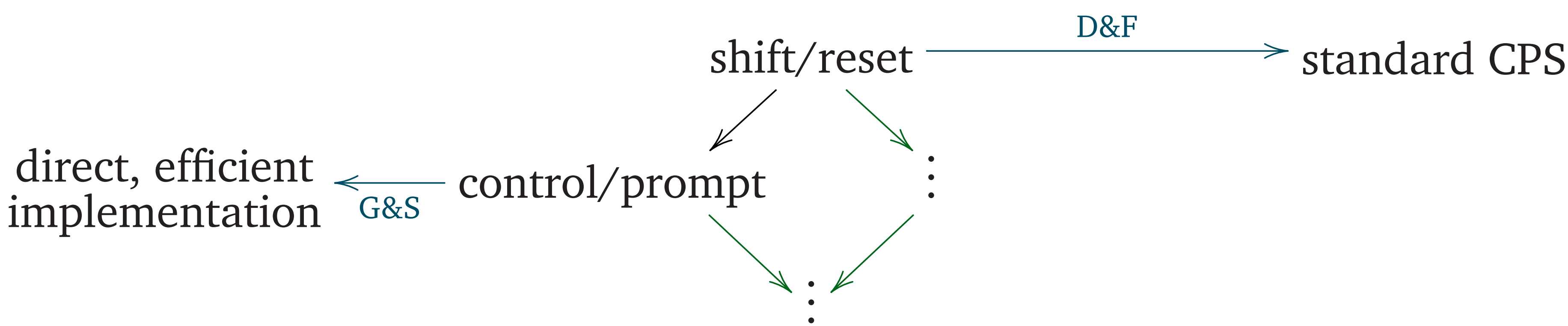
Danvy and Filinski treat shift/reset by extending a standard CPS transform.

$$\begin{tikzcd}
 & & \text{shift/reset} \arrow[r, "\text{D\&F}"] \arrow[dl] & \text{standard CPS} \\
\text{direct, efficient implementation} & \text{control/prompt} \arrow[l, "\text{G\&S}"'] & &
\end{tikzcd}$$

This paper answers two related questions.

Gasbichler and Sperber implement shift/reset and control/prompt directly.
But what about other operators?

Danvy and Filinski treat shift/reset by extending a standard CPS transform.
But what about other operators?

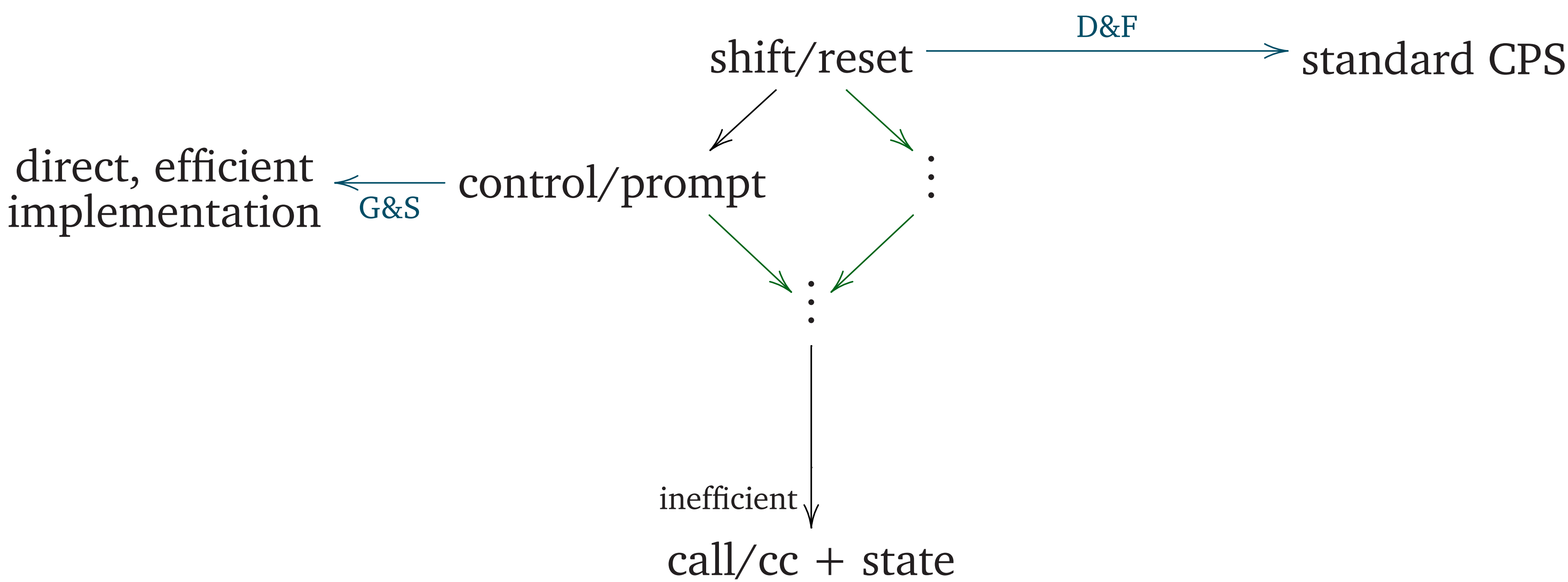


This paper answers two related questions.

Gasbichler and Sperber implement shift/reset and control/prompt directly.
But what about other operators?

- Translating to call/cc + state appears slower (Gasbichler and Sperber) and leaks memory.
- Translating to shift/reset + control/prompt is not obvious.

Danvy and Filinski treat shift/reset by extending a standard CPS transform.
But what about other operators?



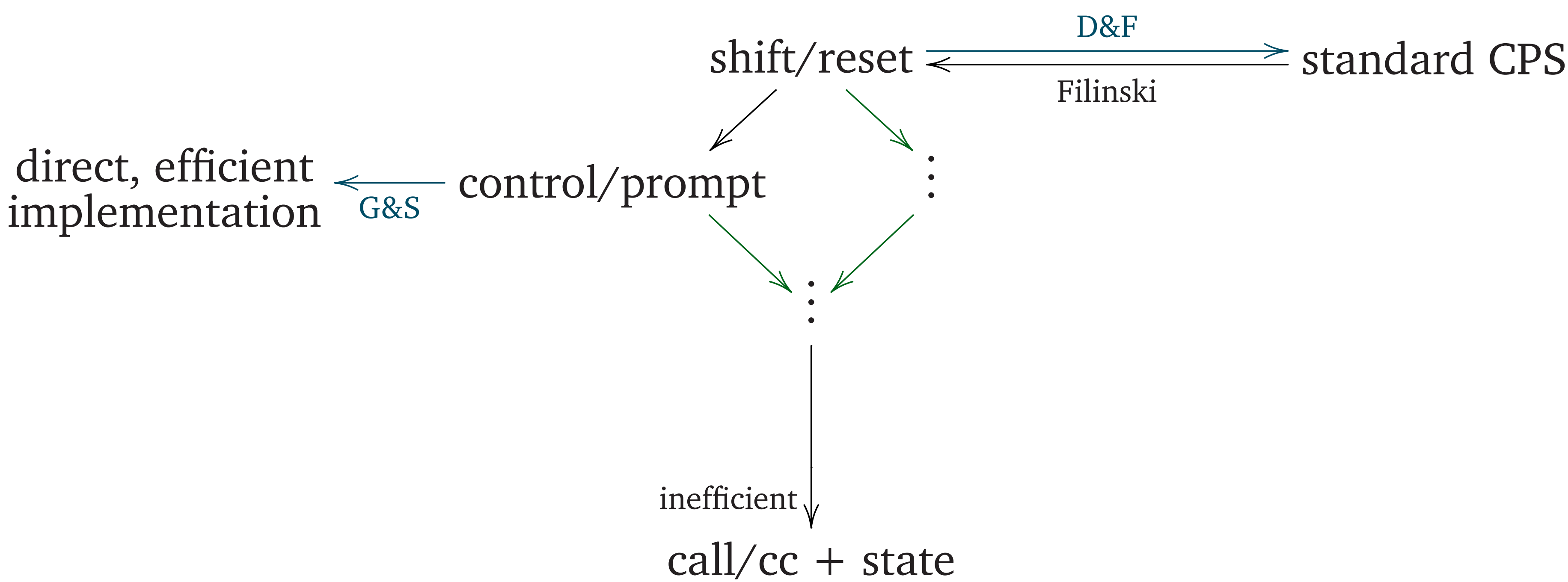
This paper answers two related questions.

Gasbichler and Sperber implement shift/reset and control/prompt directly.
But what about other operators?

- Translating to call/cc + state appears slower (Gasbichler and Sperber) and leaks memory.
- Translating to shift/reset + control/prompt is not obvious.

Danvy and Filinski treat shift/reset by extending a standard CPS transform.
But what about other operators?

- Other ("dynamic") operators are treated using activation frames, as if by necessity.
- But doesn't CPS express roughly all monads (Filinski)?



This paper answers two related questions.

Gasbichler and Sperber implement shift/reset and control/prompt directly.
But what about other operators?

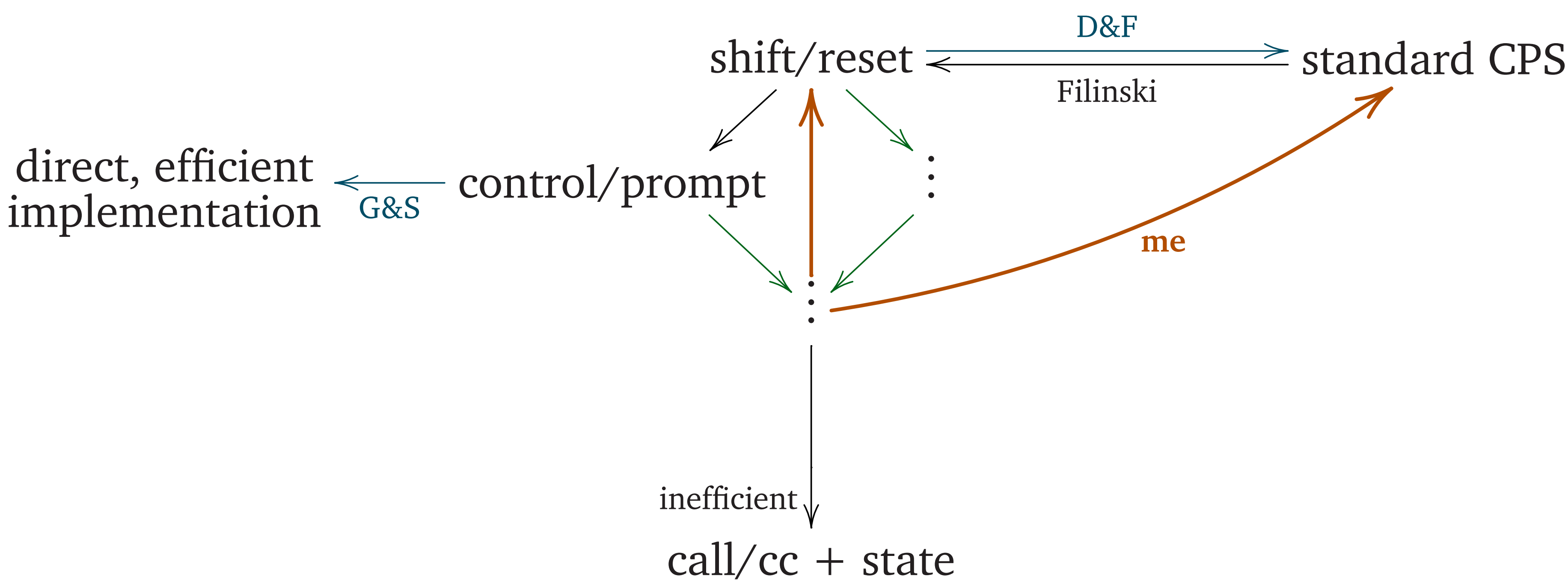
- Translating to call/cc + state appears slower (Gasbichler and Sperber) and leaks memory.
- Translating to shift/reset + control/prompt is not obvious.

It turns out that we *can* translate other operators to shift/reset!

Danvy and Filinski treat shift/reset by extending a standard CPS transform.
But what about other operators?

- Other ("dynamic") operators are treated using activation frames, as if by necessity.
- But doesn't CPS express roughly all monads (Filinski)?

It turns out that we *can* treat other operators by extending standard CPS!



This paper answers two related questions.

Gasbichler and Sperber implement shift/reset and control/prompt directly.

But what about other operators?

- Translating to call/cc + state appears slower (Gasbichler and Sperber) and leaks memory.
- Translating to shift/reset + control/ prompt is not obvious.

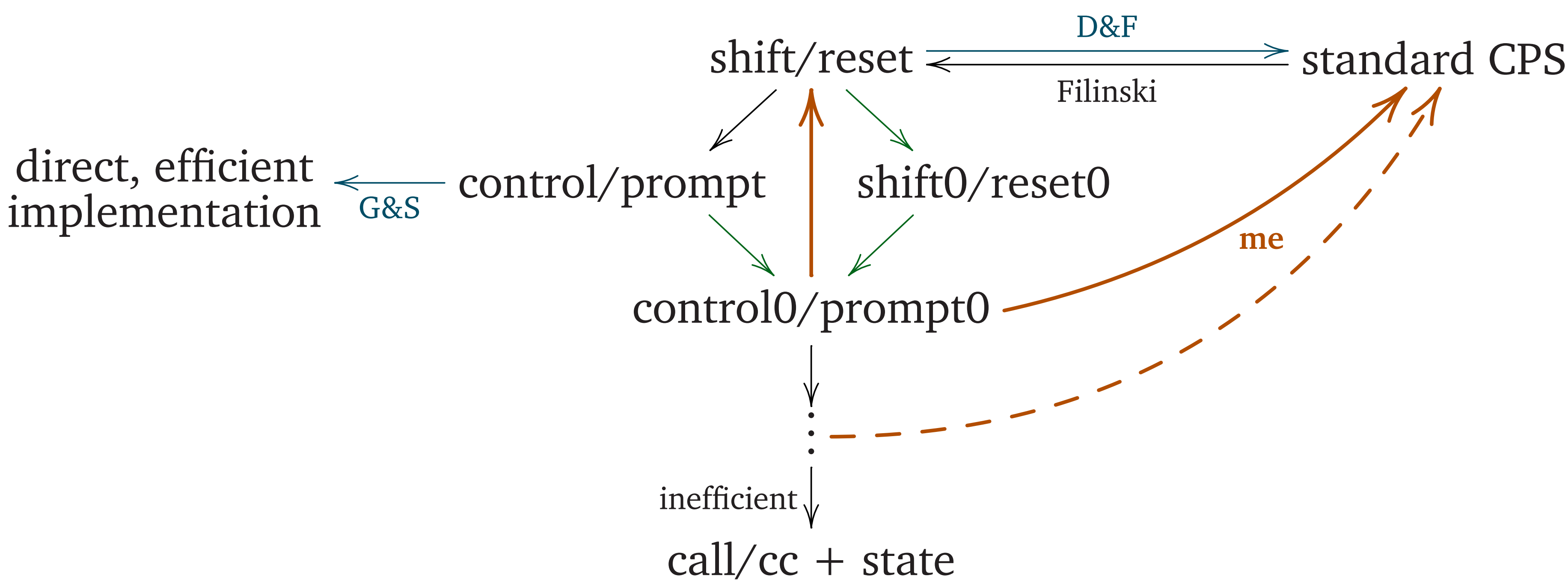
It turns out that we *can* translate other operators to shift/reset!

Danvy and Filinski treat shift/reset by extending a standard CPS transform.

But what about other operators?

- Other ("dynamic") operators are treated using activation frames, as if by necessity.
- But doesn't CPS express roughly all monads (Filinski)?

It turns out that we *can* treat other operators by extending standard CPS!



This paper answers two related questions.

Gasbichler and Sperber implement shift/reset and control/prompt directly.
But what about other operators?

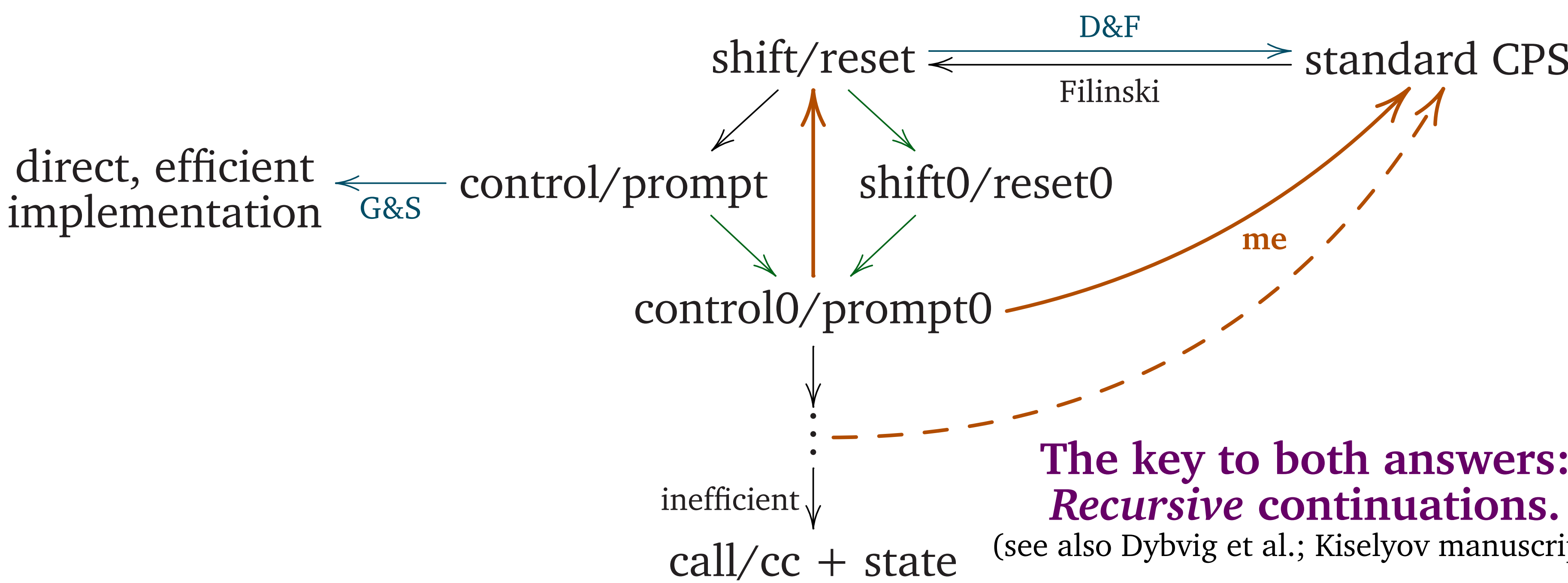
- Translating to call/cc + state appears slower (Gasbichler and Sperber) and leaks memory.
- Translating to shift/reset + control/prompt is not obvious.

It turns out that we *can* translate other operators to shift/reset!

Danvy and Filinski treat shift/reset by extending a standard CPS transform.
But what about other operators?

- Other ("dynamic") operators are treated using activation frames, as if by necessity.
- But doesn't CPS express roughly all monads (Filinski)?

It turns out that we *can* treat other operators by extending standard CPS!



The key to both answers:
***Recursive* continuations.**
(see also Dybvig et al.; Kiselyov manuscripts)

This paper answers two related questions.

Gasbichler and Sperber implement shift/reset and control/prompt directly.
But what about other operators?

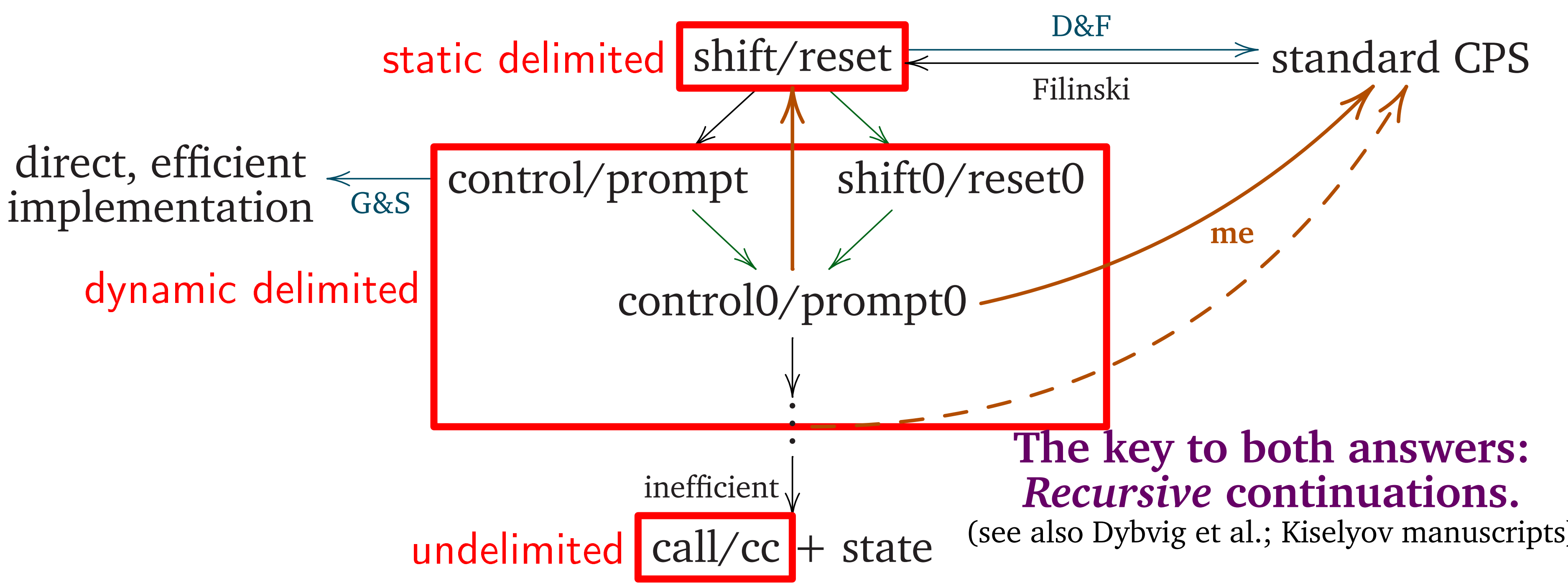
- Translating to call/cc + state appears slower (Gasbichler and Sperber) and leaks memory.
- Translating to shift/reset + control/prompt is not obvious.

It turns out that we *can* translate other operators to shift/reset!

Danvy and Filinski treat shift/reset by extending a standard CPS transform.
But what about other operators?

- Other ("dynamic") operators are treated using activation frames, as if by necessity.
- But doesn't CPS express roughly all monads (Filinski)?

It turns out that we *can* treat other operators by extending standard CPS!



The key to both answers:
***Recursive* continuations.**
(see also Dybvig et al.; Kiselyov manuscripts)

1 Operational semantics

`reset` by itself doesn't do much.

```
(cons 'x (reset (cons 'y '())))
(cons 'x (reset '(y)))
(cons 'x '(y))
'(x y)
```

1 Operational semantics

`reset` by itself doesn't do much.

```
(cons 'x (reset (cons 'y '())))
(cons 'x (reset '(y)))
(cons 'x '(y))
'(x y)
```

`shift` captures the surrounding context up to the nearest `reset`.

```
(cons 'x (reset (cons (shift c (cons 'z (c 'y))) '())))
```

1 Operational semantics

`reset` by itself doesn't do much.

```
(cons 'x (reset (cons 'y '())))
(cons 'x (reset '(y)))
(cons 'x '(y))
'(x y)
```

`shift` captures the surrounding context up to the nearest `reset`.

```
(cons 'x (reset (cons (shift c (cons 'z (c 'y))) '())))

          c ← (lambda (v) (reset (cons v '())))

(cons 'x (reset (cons 'z (reset (cons 'y '())))))
(cons 'x (reset (cons 'z (reset '(y)))))
(cons 'x (reset (cons 'z '(y))))
(cons 'x (reset '(z y)))
(cons 'x '(z y))
'(x z y)
```

1 Operational semantics

`reset` by itself doesn't do much.

```
(cons 'x (reset (cons 'y '())))
(cons 'x (reset '(y)))
(cons 'x '(y))
'(x y)
```

`shift` captures the surrounding context up to the nearest `reset`.

```
(cons 'x (reset (cons (shift c (cons 'z (c 'y))) '())))

            c ← (lambda (v) (reset (cons v '())))

(cons 'x (reset (cons 'z (reset (cons 'y '())))))
(cons 'x (reset (cons 'z (reset '(y)))))
(cons 'x (reset (cons 'z '(y))))
(cons 'x (reset '(z y)))
(cons 'x '(z y))
'(x z y)
```

Note how the captured context `(cons □ '())` is represented as a function `c`.

(Note also how `shift` keeps the delimiting `reset`.)

The fine print matters when the captured context itself contains `shift`.

```
(cons 'x (reset (cons (shift c (cons 'z (c 'y))) (shift d '()))))
```

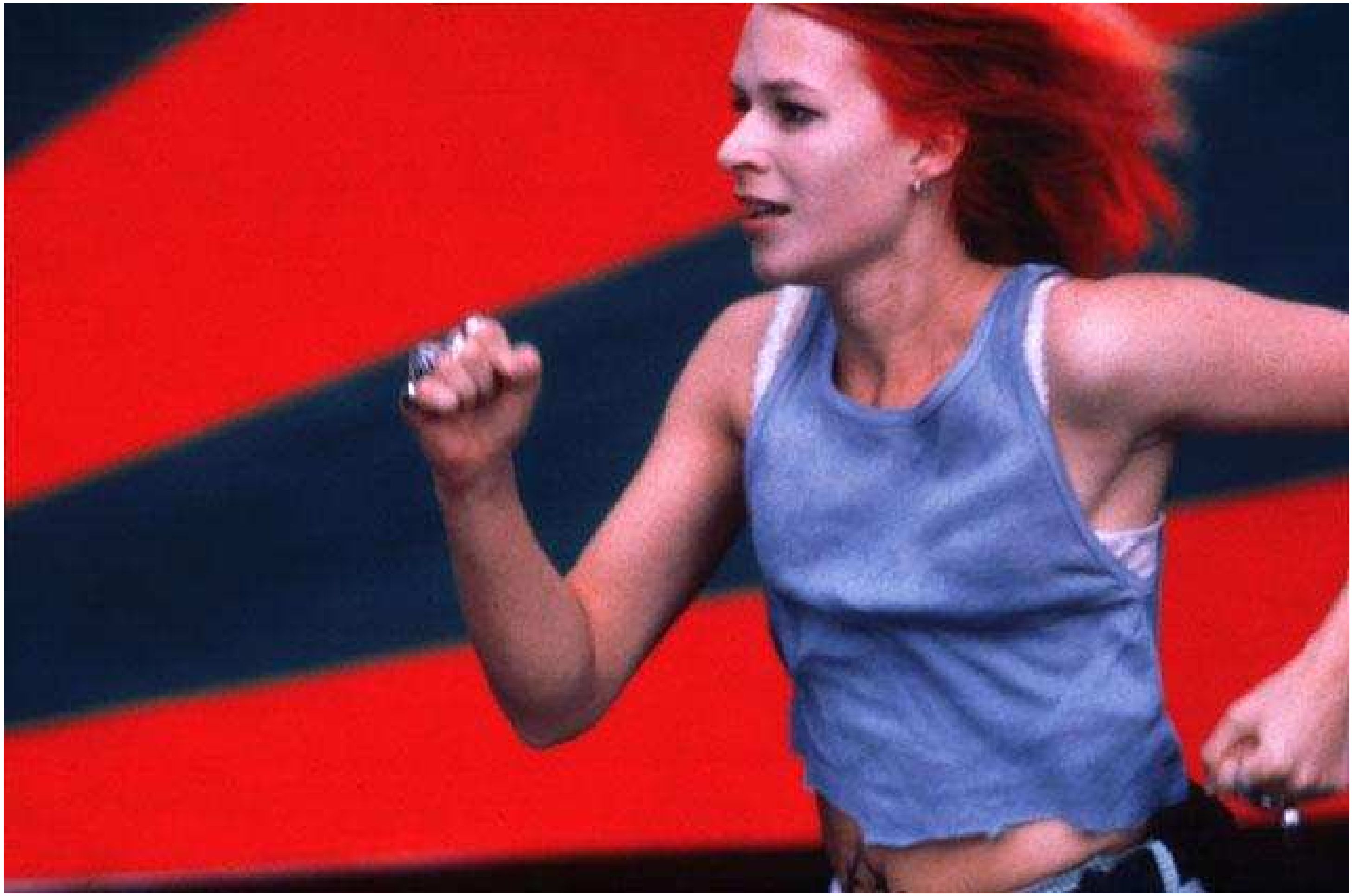
The fine print matters when the captured context itself contains `shift`.

```
(cons 'x (reset (cons (shift c (cons 'z (c 'y))) (shift d '()))))

           c ← (lambda (v) (reset (cons v (shift d '()))))

(cons 'x (reset (cons 'z (reset (cons 'y (shift d '()))))))
(cons 'x (reset (cons 'z (reset '()))))
(cons 'x (reset (cons 'z '())))
(cons 'x (reset '(z)))
(cons 'x '(z))
'(x z)
```

The fine print matters when the captured context itself contains `shift`.

```
(cons 'x (reset (cons (shift c (cons 'z (c 'y))) (shift d '()))))
```

c ← (lambda (v) ~~(reset~~ (cons v (shift d '()))~~)~~)



The fine print matters when the captured context itself contains `shift`.

```
(cons 'x (reset (cons (shift c (cons 'z (c 'y))) (shift d '()))))

             c ⟵ (lambda (v)          (cons v (shift d '())) )

(cons 'x (reset (cons 'z (cons 'y (shift d '())))))
(cons 'x (reset '()))
(cons 'x '())
'(x)
```

The fine print matters when the captured context itself contains `control`.

```
(cons 'x (prompt (cons (control c (cons 'z (c 'y))) (control d '()))))

            c ← (lambda (v)       (cons v (control d '())) )

(cons 'x (prompt (cons 'z (cons 'y (control d '())))))
(cons 'x (prompt '()))
(cons 'x '())
'(x)
```

That `shift`/`reset` is called `control`/`prompt`.

Contexts captured by `shift` are always pure.
Contexts captured by `control` are not.

It is obvious how to translate `shift`/`reset` to `control`/`prompt`.
"The other direction, though, seems not to be known."

(The other fine print matters when the `shift` expression's body `shifts` again.)

2 CPS transform

The CPS transform allows an expression to manipulate its own context by passing the context (continuation) as an argument.

$$\overline{\texttt{x}} = \texttt{(lambda (c) (c x))}$$

$$\overline{\texttt{(lambda (x) E)}} = \texttt{(lambda (c) (c (lambda (x) }\overline{\texttt{E}}\texttt{)))}$$

$$\begin{aligned}\overline{\texttt{(E}_1\texttt{ E}_2\texttt{)}} = \texttt{(lambda (c) (}&\overline{\texttt{E}_1}\texttt{ (lambda (f)}\\ &\qquad\texttt{(}\overline{\texttt{E}_2}\texttt{ (lambda (x) ((f x) c))))))}\end{aligned}$$

$$\overline{\texttt{(reset E)}} = \texttt{(lambda (c) (c (}\overline{\texttt{E}}\texttt{ (lambda (v) v))))}$$

$$\begin{aligned}\overline{\texttt{(shift f E)}} = \texttt{(lambda (c) (let ((f (lambda (x)}\\ \texttt{(lambda (c2) (c2 (c x))))))}\\ \texttt{(}\overline{\texttt{E}}\texttt{ (lambda (v) v))))}\end{aligned}$$

$$\overline{\texttt{(prompt E)}} = ???$$

$$\overline{\texttt{(control f E)}} = ???$$

Key idea: pass the context as an argument.

For `shift/reset`, captured contexts are always pure, so we can just represent

$$\text{contexts} \quad = \quad \text{values} \to \text{answers}.$$

For `control/prompt`, a captured context may also manipulate its own invocation contexts, so we need to pass the invocation contexts as an additional argument.

$$\begin{aligned}\text{contexts} \quad &= \quad (\text{values} \times \text{contexts}) \to \text{answers} \\ &= \quad \text{values} \to (\text{contexts} \to \text{answers}).\end{aligned}$$

Recursion!

The empty context is not the identity anymore: the context argument is new.

$$\boxed{\text{contexts} \quad = \quad \text{values} \rightarrow (\text{contexts} \rightarrow \text{answers})}$$

The empty context `send`: `context[send[value]]`

```
(define send
  (lambda (value)
    (lambda (context)
      (if (eq? context send)
          value
          ((context value) send)))))
```

To concatenate two contexts: `context[d[c[value]]]`

```
(define (compose c d)
  (if (eq? d send)
      c
      (lambda (value)
        (lambda (context)
          ((c value) (compose d context))))))
```

(We use `eq?` to intensionally compare contexts to `send`,
so the type above $\text{values} \rightarrow (\text{contexts} \rightarrow \text{answers})$
should really be $\text{values} \rightarrow (\text{contexts option} \rightarrow \text{answers})$.)

$$\boxed{\text{contexts} \quad = \quad \text{values} \to (\text{contexts} \to \text{answers})}$$

How to transform expressions?

The standard CPS transform produces types of the form

$$\underbrace{(A \to W)}_{\text{context}} \to W.$$

By unification, let $A = \text{values}$, $W = \text{contexts} \to \text{answers}$.

$$\text{computations} \quad = \quad (\text{values} \to \text{contexts} \to \text{answers}) \to \text{contexts} \to \text{answers}$$

Almost done!

(In Danvy and Filinski's terminology, dynamic control is recursive because it unifies contexts with metacontexts.)

$$
\boxed{\begin{array}{rcl}
\text{contexts} & = & \text{values} \to \text{contexts} \to \text{answers} \\
\text{computations} & = & (\text{values} \to \text{contexts} \to \text{answers}) \to \text{contexts} \to \text{answers}
\end{array}}
$$

$$
\begin{array}{rcl}
\text{contexts} & = & \text{values} \to \text{contexts} \to \text{answers} \\
\text{computations} & = & (\text{values} \to \text{contexts} \to \text{answers}) \to \text{contexts} \to \text{answers}
\end{array}
$$

$\overline{\texttt{(prompt E)}} =$

```
(lambda (c) (c ((Ē send) send)))
```

$\overline{\texttt{(control f E)}} =$

```
(lambda (c1) (lambda (d1)
  (let ((f (lambda (v) (lambda (c2) (lambda (d2)
                (((compose c1 d1) v) (compose c2 d2)))))))
    ((Ē send) send))))
```

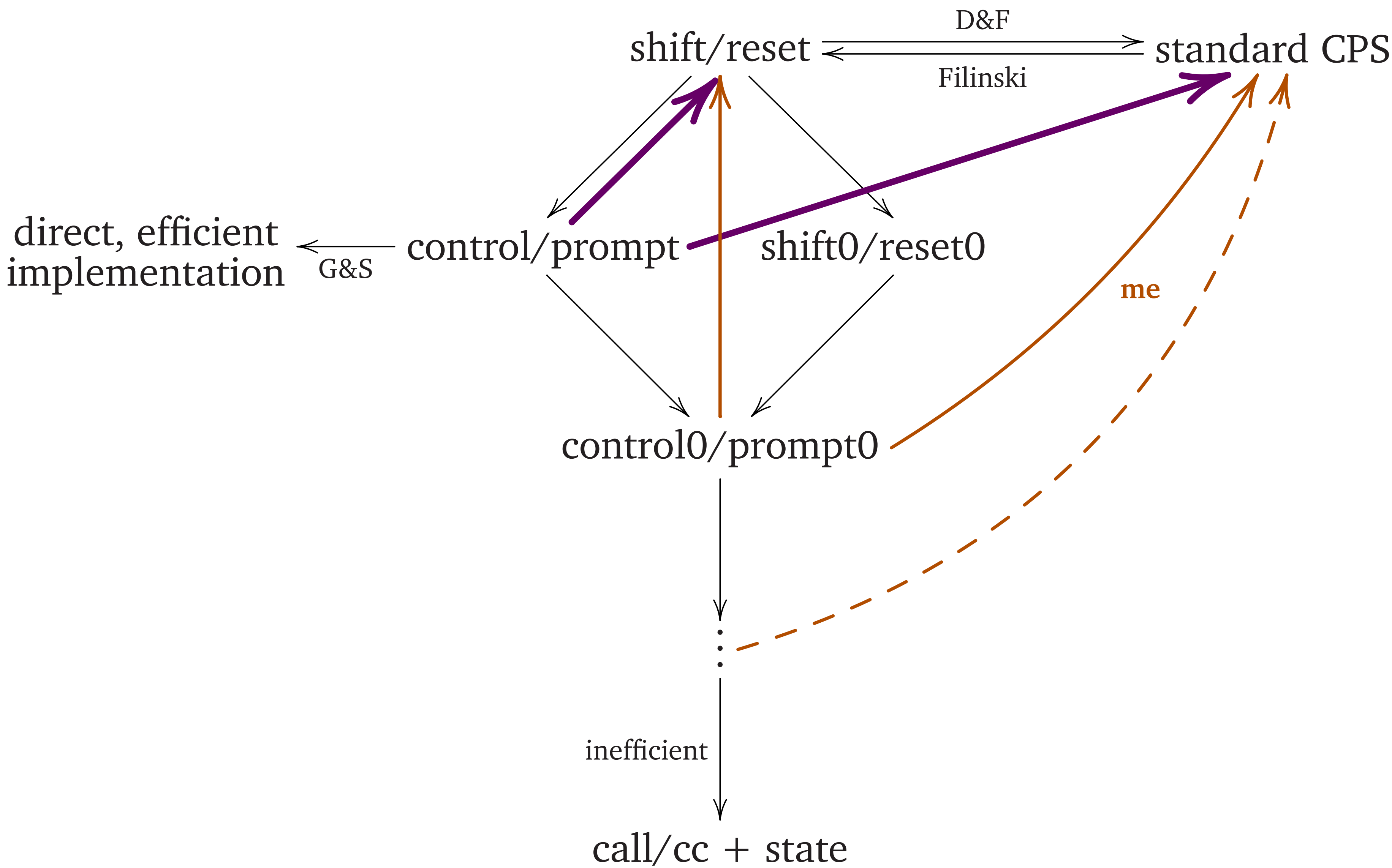
contexts	=	values $\rightarrow$ contexts $\rightarrow$ answers
computations	=	(values $\rightarrow$ contexts $\rightarrow$ answers) $\rightarrow$ contexts $\rightarrow$ answers

$$\overline{\texttt{(prompt E)}} =$$
$$\texttt{(lambda (c) (c ((}\overline{\texttt{E}}\texttt{ send) send)))}$$

$$\overline{\texttt{(control f E)}} =$$
$$\begin{array}{l}
\texttt{(lambda (c1) (lambda (d1)} \\
\quad\texttt{(let ((f (lambda (v) (lambda (c2) (lambda (d2)}} \\
\qquad\qquad\texttt{(((compose c1 d1) v) (compose c2 d2)))))))} \\
\quad\;\;\texttt{((}\overline{\texttt{E}}\texttt{ send) send))))}
\end{array}$$

```
(define-syntax prompt
  (syntax-rules ()
    ((_ e) ((reset (send e)) send))))

(define-syntax control
  (syntax-rules ()
    ((_ f e)
     (shift c1 (lambda (d1)
       (let ((f (lambda (v) (shift c2 (lambda (d2)
                   (((compose c1 d1) v) (compose c2 d2)))))))
         ((reset (send e)) send)))))))
```

3 Summary



***Dynamic* delimited continuations are *recursive* delimited continuations.**