Continuations and Transducer Composition

Olin Shivers Matthew Might

Georgia Tech

PLDI 2006

The Big Idea

Observation

Some programs easier to write with transducer abstraction.

Goal

Design features and compilation story to support this abstraction.

The Big Idea

Observation

Some programs easier to write with transducer abstraction.

Goal

Design features and compilation story to support this abstraction.

Oh...

Transducer $\equiv$ Coroutine $\equiv$ Process

A computational analogy

The world of functions

- Agents are functions.
- Functions are stateless.
- Composed with $\circ$ operator: $h = f \circ g$.

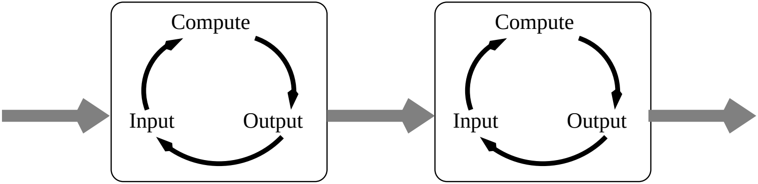
A computational analogy

The world of functions

- Agents are functions.
- Functions are stateless.
- Composed with $\circ$ operator: $h = f \circ g$.

The world of online transducers

- Agents are input/compute/output processes.
- Processes have local, bounded state.
- Composed with Unix | operator: $h = g \mid f$.



Online transducers

- DSP networks
  Convolve / integrate / filter / difference / . . .
- Network-protocol stacks (“micro-protocols”, layer integration)
  packet-assembly / checksum / order / http-parse / html-lex / . . .
- Graphics processing
  viewpoint-transform / clip1 / . . . / clip6 / z-divide / light / scan
- Stream processing
- Unix pipelines

⋮

Optimisation across composition

Functional paradigm

$f \circ g$ optimised by β-reduction:

$$f = \lambda y \,.\, y + 3$$
$$g = \lambda z \,.\, z + 5$$

Optimisation across composition

Functional paradigm

$f \circ g$ optimised by β-reduction:

$$f = \lambda y \,.\, y + 3$$
$$g = \lambda z \,.\, z + 5$$
$$\circ = \lambda m\, n \,.\, \lambda x . m(n\, x)$$

(“Plumbing” made explicit in λ rep.)

Optimisation across composition

Functional paradigm

$f \circ g$ optimised by β-reduction:

$$f = \lambda y \,.\, y + 3$$
$$g = \lambda z \,.\, z + 5$$
$$\circ = \lambda m\, n \,.\, \lambda x.m(n\, x)$$

("Plumbing" made explicit in λ rep.)

$$f \circ g = (\lambda mn.\lambda x.m(nx))(\lambda y.y + 3)(\lambda z.z + 5)$$

Optimisation across composition

Functional paradigm

$f \circ g$ optimised by β-reduction:

$$f = \lambda y\,.\,y + 3$$
$$g = \lambda z\,.\,z + 5$$
$$\circ = \lambda m\,n\,.\,\lambda x.m(n\,x)$$

(“Plumbing” made explicit in λ rep.)

$$\begin{aligned}
f \circ g &= (\lambda mn.\lambda x.m(nx))(\lambda y.y + 3)(\lambda z.z + 5) \\
&= \lambda x.(\lambda y.y + 3)((\lambda z.z + 5)x) \\
&= \lambda x.(\lambda y.y + 3)(x + 5) \\
&= \lambda x.(x + 5) + 3 \\
&= \lambda x.x + (5 + 3) \\
&= \lambda x.x + 8
\end{aligned}$$

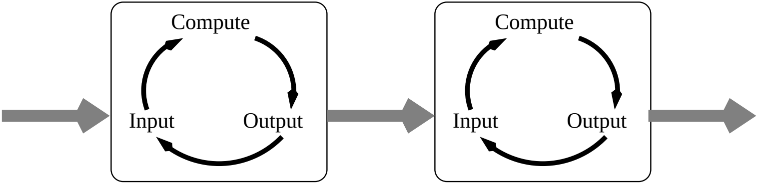
Optimisation across composition

Transducer paradigm

No good optimisation story.

Optimisation across composition is
key technology supporting abstraction:
Enables construction by composition.

If only. . .



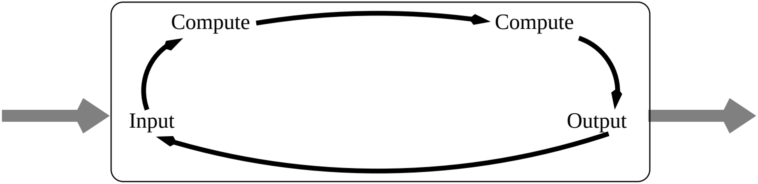
Optimisation across composition

Transducer paradigm

No good optimisation story.

Optimisation across composition is
key technology supporting abstraction:
Enables construction by composition.

If only. . .



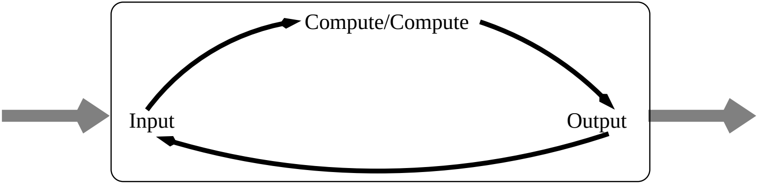
Optimisation across composition

Transducer paradigm

No good optimisation story.

Optimisation across composition is
key technology supporting abstraction:
Enables construction by composition.

If only. . .



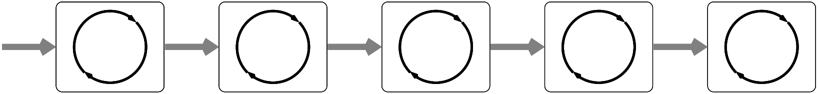
Optimisation across composition

Transducer paradigm

No good optimisation story.

Optimisation across composition is
key technology supporting abstraction:
Enables construction by composition.

If only...



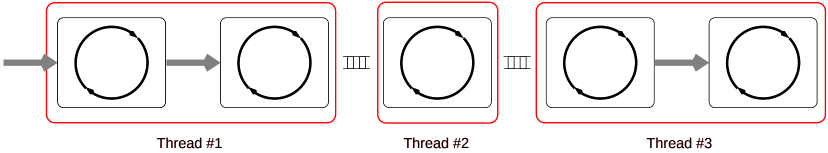
Optimisation across composition

Transducer paradigm

No good optimisation story.

Optimisation across composition is
key technology supporting abstraction:
Enables construction by composition.

If only. . .



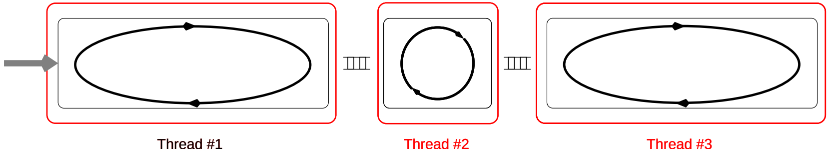
Optimisation across composition

Transducer paradigm

No good optimisation story.

Optimisation across composition is
key technology supporting abstraction:
Enables construction by composition.

If only. . .



Strategy

- Build transducers from continuations.

Strategy

- Build transducers from continuations.
- Build continuations from λ.

Strategy

- Build transducers from continuations.
- Build continuations from λ.
- Handle λ well.

Strategy

- Build transducers from continuations.
- Build continuations from λ.
- Handle λ well.
- Watch what happens.

Tool: Continuation-passing style (CPS)

Restricted subset of λ calculus: Function calls do not return.

Thus cannot write `f(g(x))`.

Must pass extra argument—the *continuation*—to each call,
to represent rest of computation:

`(- a (* b c))` $\Rightarrow$ `(* b c (λ (temp) (- a temp halt)))`

Tool: Continuation-passing style (CPS)

Restricted subset of λ calculus: Function calls do not return.

Thus cannot write `f(g(x))`.

Must pass extra argument—the *continuation*—to each call,
to represent rest of computation:

`(- a (* b c))` $\Rightarrow$ `(* b c (λ (temp) (- a temp halt)))`

CPS is the “assembler” of functional languages.

CPS Payoff

CPS is universal representation of control & env.

Construct	encoding
fun call	call to λ

CPS Payoff

CPS is universal representation of control & env.

Construct	encoding
fun call	call to λ
fun return	call to λ

CPS Payoff

CPS is universal representation of control & env.

Construct	encoding
fun call	call to λ
fun return	call to λ
iteration	call to λ

CPS Payoff

CPS is universal representation of control & env.

Construct	encoding
fun call	call to λ
fun return	call to λ
iteration	call to λ
sequencing	call to λ

CPS Payoff

CPS is universal representation of control & env.

Construct	encoding
fun call	call to λ
fun return	call to λ
iteration	call to λ
sequencing	call to λ
conditional	call to λ

CPS Payoff

CPS is universal representation of control & env.

Construct	encoding
fun call	call to λ
fun return	call to λ
iteration	call to λ
sequencing	call to λ
conditional	call to λ
exception	call to λ

CPS Payoff

CPS is universal representation of control & env.

Construct	encoding
fun call	call to λ
fun return	call to λ
iteration	call to λ
sequencing	call to λ
conditional	call to λ
exception	call to λ
continuation	call to λ

CPS Payoff

CPS is universal representation of control & env.

Construct	encoding
fun call	call to λ
fun return	call to λ
iteration	call to λ
sequencing	call to λ
conditional	call to λ
exception	call to λ
continuation	call to λ
coroutine switch	call to λ
$\vdots$	$\vdots$

Writing transducers with `put` and `get`

```
(define (send-fives)
  (put 5)
  (send-fives))
```

Writing transducers with `put` and `get`

```
(define (send-fives)
  (put 5)
  (send-fives))

(define (doubler)
  (put (* 2 (get)))
  (doubler))
```

Writing transducers with put and get

```
(define (send-fives)
  (put 5)
  (send-fives))

(define (doubler)
  (put (* 2 (get)))
  (doubler))

(define (integ sum)
  (let ((next-sum (+ sum (get))))
    (put next-sum)
    (integ next-sum)))
```

Tool: 3CPS & transducer pipelines

$$f\ x\ k\ u\ d$$

Tool: 3CPS & transducer pipelines

$$f\ x\ k\ u\ d$$

ExpCont: rest of this stage's computation

Semantic domains / Types

$$x \in \text{Value}$$
$$k \in \text{ExpCont} \quad = \text{Value} \rightarrow \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans}$$

(a transducer)

Tool: 3CPS & transducer pipelines

$f\ x\ k\ u\ d$

ExpCont: rest of this stage's computation

UpCont: rest of upstream computation

Semantic domains / Types

$$
\begin{aligned}
x &\in \text{Value} & & \\
k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans}
\end{aligned}
$$

(a transducer)

Tool: 3CPS & transducer pipelines

$f\,x\,k\,u\,d$

- ExpCont: rest of this stage's computation
- UpCont: rest of upstream computation
- DownCont: rest of downstream computation

Semantic domains / Types

$$
\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans} \\
d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans} \\
c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans} \text{ (a transducer)}
\end{aligned}
$$

Transducers in 3CPS

Get & put in 3CPS

$$
\begin{aligned}
get\; x\; k\; u\; d &= \\
put\; x\; k\; u\; d &=
\end{aligned}
$$

Semantic domains / Types

$$
\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} &&= \text{Value} \rightarrow \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \rightarrow \text{Ans} \\
d &\in \text{DownCont} &&= \text{Value} \rightarrow \text{UpCont} \rightarrow \text{Ans} \\
c &\in \text{CmdCont} &&= \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans}
\end{aligned}
$$

Transducers in 3CPS

Get & put in 3CPS

$$\begin{aligned} get\, x\, k\, u\, d &= u\, (\qquad\qquad\qquad\qquad) \\ put\, x\, k\, u\, d &= \end{aligned}$$

Semantic domains / Types

$$\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans} \\
d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans} \\
c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans}
\end{aligned}$$

Transducers in 3CPS

Get & put in 3CPS

$$\begin{aligned} get\, x\, k\, u\, d &= u\, (\lambda\, x'\, u' \,. \qquad\qquad) \\ put\, x\, k\, u\, d &= \end{aligned}$$

Semantic domains / Types

$$\begin{aligned} x &\in \text{Value} \\ k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\ u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans} \\ d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans} \\ c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans} \end{aligned}$$

Transducers in 3CPS

Get & put in 3CPS

$$get\ x\ k\ u\ d = u\ (\lambda\, x'\ u'\ .\ k \qquad\qquad)$$
$$put\ x\ k\ u\ d =$$

Semantic domains / Types

$$\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} = \text{Value} \rightarrow \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans} \\
u &\in \text{UpCont} = \text{DownCont} \rightarrow \text{Ans} \\
d &\in \text{DownCont} = \text{Value} \rightarrow \text{UpCont} \rightarrow \text{Ans} \\
c &\in \text{CmdCont} = \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans}
\end{aligned}$$

Transducers in 3CPS

Get & put in 3CPS

$$\begin{aligned} get\,x\,k\,u\,d &= u\,(\lambda\,x'\,u'\,.\,k\,x' \qquad) \\ put\,x\,k\,u\,d &= \end{aligned}$$

Semantic domains / Types

$$\begin{aligned} x &\in \text{Value} \\ k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\ u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans} \\ d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans} \\ c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans} \end{aligned}$$

Transducers in 3CPS

Get & put in 3CPS

$$get\; x\; k\; u\; d = u\; (\lambda\, x'\; u'\; .\; k\; x'\; u'\; d)$$
$$put\; x\; k\; u\; d =$$

Semantic domains / Types

$$
\begin{aligned}
x &\in \text{Value} \\
{\color{red} k} &\;{\color{red}\in \text{ExpCont}} && {\color{red}= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans}} \\
u &\in \text{UpCont} && = \text{DownCont} \to \text{Ans} \\
d &\in \text{DownCont} && = \text{Value} \to \text{UpCont} \to \text{Ans} \\
c &\in \text{CmdCont} && = \text{UpCont} \to \text{DownCont} \to \text{Ans}
\end{aligned}
$$

Transducers in 3CPS

Get & put in 3CPS

$$\begin{aligned} get\,x\,k\,u\,d &= u\,(\lambda\,x'\,u'\,.\,k\ x'\ u'\ d) \\ put\,x\,k\,u\,d &= d \end{aligned}$$

Semantic domains / Types

$$\begin{aligned} x &\in \text{Value} \\ k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\ u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans} \\ d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans} \\ c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans} \end{aligned}$$

Transducers in 3CPS

Get & put in 3CPS

$$
\begin{aligned}
get\,x\,k\,u\,d &= u\,(\lambda\,x'\,u'\,.\,k\,x'\,u'\,d)\\
put\,x\,k\,u\,d &= d\,x\,(\qquad\qquad\qquad\qquad)
\end{aligned}
$$

Semantic domains / Types

$$
\begin{aligned}
x &\in \text{Value}\\
k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans}\\
u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans}\\
d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans}\\
c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans}
\end{aligned}
$$

Transducers in 3CPS

Get & put in 3CPS

$$
\begin{aligned}
get\,x\,k\,u\,d &= u\,(\lambda\,x'\,u'\,.\,k\;x'\;u'\;d)\\
put\,x\,k\,u\,d &= d\;x\;(\lambda\,d'\,.\qquad\qquad\qquad)
\end{aligned}
$$

Semantic domains / Types

$$
\begin{aligned}
x &\in \text{Value}\\
k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans}\\
u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans}\\
d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans}\\
c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans}
\end{aligned}
$$

Transducers in 3CPS

Get & put in 3CPS

$$\begin{aligned} get\,x\,k\,u\,d &= u\,(\lambda\,x'\,u'\,.\,k\,x'\,u'\,d) \\ put\,x\,k\,u\,d &= d\,x\,(\lambda\,d'\,.\,k\,unit\qquad) \end{aligned}$$

Semantic domains / Types

$$\begin{aligned} x &\in \text{Value} \\ k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\ u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans} \\ d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans} \\ c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans} \end{aligned}$$

Transducers in 3CPS

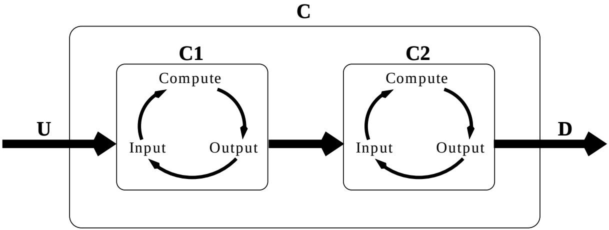
Get & put in 3CPS

$$\begin{aligned} get\; x\; k\; u\; d &= u\;(\lambda\, x'\; u'\;.\; k\; x'\; u'\; d) \\ put\; x\; k\; u\; d &= d\; x\;(\lambda\, d'\;.\; k\; unit\; u\; d') \end{aligned}$$

Semantic domains / Types

$$\begin{aligned} x &\in \text{Value} \\ {\color{red}k} &{\color{red}\in \text{ExpCont}} && {\color{red}= \text{Value} \rightarrow \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans}} \\ u &\in \text{UpCont} && = \text{DownCont} \rightarrow \text{Ans} \\ d &\in \text{DownCont} && = \text{Value} \rightarrow \text{UpCont} \rightarrow \text{Ans} \\ c &\in \text{CmdCont} && = \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans} \end{aligned}$$

Composing transducers in 3CPS

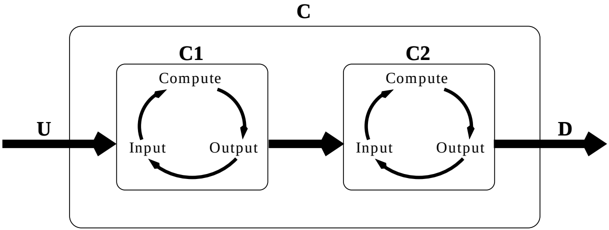


$$compose/pull \;\; c_1 \;\; c_2$$

Semantic domains / Types

$$\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans} \\
d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans} \\
c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans}
\end{aligned}$$

Composing transducers in 3CPS

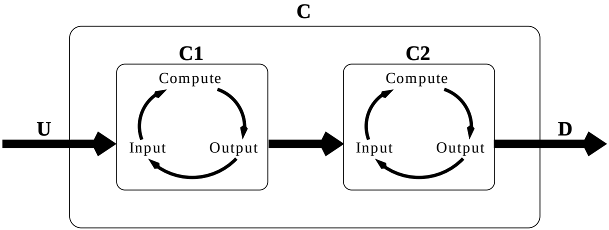


$$\textit{compose/pull}\ \ c_1\ c_2 = \lambda\, u\, d\ .$$

Semantic domains / Types

$$
\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans} \\
d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans} \\
c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans}
\end{aligned}
$$

Composing transducers in 3CPS

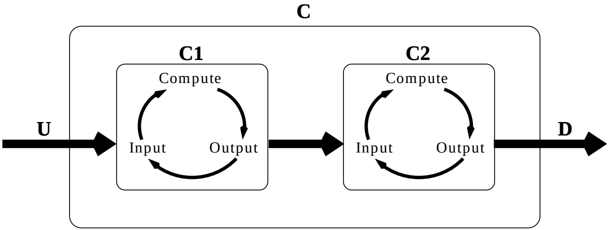


$$compose/pull\;\; c_1\; c_2 = \lambda\, u\, d\; .\; c_2$$

Semantic domains / Types

$$\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} &&= \text{Value} \rightarrow \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \rightarrow \text{Ans} \\
d &\in \text{DownCont} &&= \text{Value} \rightarrow \text{UpCont} \rightarrow \text{Ans} \\
c &\in \text{CmdCont} &&= \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans}
\end{aligned}$$

Composing transducers in 3CPS

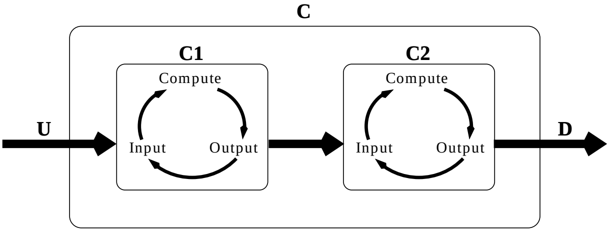


$$compose/pull\ \ c_1\ c_2 = \lambda\, u\, d\ .\ c_2\,(\lambda\, d'\ .\qquad\qquad)\, d$$

Semantic domains / Types

$$
\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} &&= \text{Value} \rightarrow \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \rightarrow \text{Ans} \\
d &\in \text{DownCont} &&= \text{Value} \rightarrow \text{UpCont} \rightarrow \text{Ans} \\
c &\in \text{CmdCont} &&= \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans}
\end{aligned}
$$

Composing transducers in 3CPS

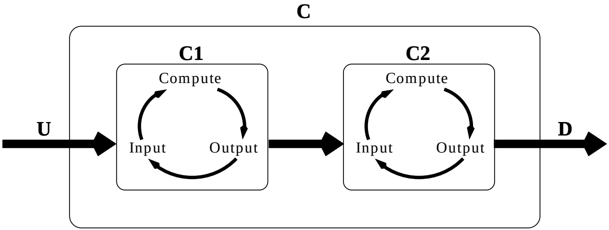


$$compose/pull\ \ c_1\ c_2 = \lambda\, u\, d\ .\ c_2\, (\lambda\, d'\ .\ c_1 \qquad)\, d$$

Semantic domains / Types

$$\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} &&= \text{Value} \rightarrow \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \rightarrow \text{Ans} \\
d &\in \text{DownCont} &&= \text{Value} \rightarrow \text{UpCont} \rightarrow \text{Ans} \\
c &\in \text{CmdCont} &&= \text{UpCont} \rightarrow \text{DownCont} \rightarrow \text{Ans}
\end{aligned}$$

Composing transducers in 3CPS

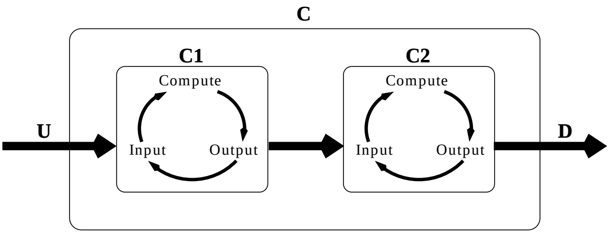


$$compose/pull\ \ c_1\ c_2 = \lambda\, u\, d\ .\ c_2\,(\lambda\, d'\ .\ c_1\, u\quad)\, d$$

Semantic domains / Types

$$\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans} \\
d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans} \\
c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans}
\end{aligned}$$

Composing transducers in 3CPS



$$compose/pull\ \ c_1\ c_2 = \lambda\, u\, d\ .\ c_2\,(\lambda\, d'\ .\ c_1\, u\, d')\, d$$

Semantic domains / Types

$$\begin{aligned}
x &\in \text{Value} \\
k &\in \text{ExpCont} &&= \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\
u &\in \text{UpCont} &&= \text{DownCont} \to \text{Ans} \\
d &\in \text{DownCont} &&= \text{Value} \to \text{UpCont} \to \text{Ans} \\
c &\in \text{CmdCont} &&= \text{UpCont} \to \text{DownCont} \to \text{Ans}
\end{aligned}$$

Transducer data/control flow in 3CPS

$$
\begin{aligned}
\mathit{get}\; x\; k\; u\; d &= u\;(\lambda\, x\; u'\;.\; k\;\; x\;\; u'\;\; d) \\
\mathit{put}\; x\; k\; u\; d &= d\;\; x\;(\lambda\, d'\;.\; k\;\; \mathit{unit}\;\; u\;\; d) \\
\mathit{compose}/\mathit{pull}\;\; c_1\;\; c_2 &= \lambda\, u\; d\;.\; c_2\,(\lambda\, d'\;.\; c_1\; u\; d')\; d
\end{aligned}
$$

Transducer data/control flow in 3CPS

$$
\begin{aligned}
\mathit{get}\, x\, k\, u\, d &= u\,(\lambda\, x\, u' \,.\, k\ x\ u'\ d) \\
\mathit{put}\, x\, k\, u\, d &= d\ x\,(\lambda\, d' \,.\, k\ \mathit{unit}\ u\ d) \\
\mathit{compose}/\mathit{pull}\ \ c_1\ c_2 &= \lambda\, u\, d \,.\, c_2\,(\lambda\, d' \,.\, c_1\, u\, d')\, d
\end{aligned}
$$

All the “plumbing” made explicit
in three short equations.

A toy example

```
(λ ()                         ; Put-5
  (letrec ((lp1 (λ () (put 5) (lp1))))
    (lp1)))
```

```
(λ ()                         ; Doubler
  (letrec ((lp2 (λ ()
                  (put (* 2 (get)))
                  (lp2))))
    (lp2)))
```

After CPS conversion

```
(λ (k1 u1 d1)                    ; Put-5
  (letrec ((lp1 (λ (k1a u1a d1a)
                  (d1a 5 (λ (d1b) (lp1 k1a u1a d1b))))))
    (lp1 k1 u1 d1)))
(λ (k2 u2 d2)                    ; Doubler
  (letrec ((lp2 (λ (k2a u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 k2a u2b d2b))))))))
    (lp2 k2 u2 d2)))
```

(compose/pull put-5 doubler)

```
((λ (c1 c2)                          ; Compose/pull
   (λ (k u d) (c2 k (λ (d') (c1 k u d')) d)))
 (λ (k1 u1 d1)                       ; Put-5
   (letrec ((lp1 (λ (k1a u1a d1a)
                   (d1a 5 (λ (d1b) (lp1 k1a u1a d1b))))))
     (lp1 k1 u1 d1)))
 (λ (k2 u2 d2)                       ; Doubler
   (letrec ((lp2 (λ (k2a u2a d2a)
                   (u2a (λ (x u2b)
                          (d2a (* 2 x)
                               (λ (d2b)
                                 (lp2 k2a u2b d2b))))))))
     (lp2 k2 u2 d2))))
```

(compose/pull put-5 doubler)

```
((λ (c1 c2)                        ; Compose/pull
   (λ (k u d) (c2 k (λ (d') (c1 k u d')) d)))
 (λ (k1 u1 d1)                     ; Put-5
   (letrec ((lp1 (λ (k1a u1a d1a)
                   (d1a 5 (λ (d1b) (lp1 k1a u1a d1b)))))))
     (lp1 k1 u1 d1)))
 (λ (k2 u2 d2)                     ; Doubler
   (letrec ((lp2 (λ (k2a u2a d2a)
                   (u2a (λ (x u2b)
                          (d2a (* 2 x)
                               (λ (d2b)
                                 (lp2 k2a u2b d2b)))))))))
     (lp2 k2 u2 d2))))
```

Eliminate useless variables (1991)

(compose/pull put-5 doubler)

```
((λ (c1 c2)                          ; Compose/pull
   (λ (k u d) (c2 (λ (d') (c1 d')) d)))
 (λ (d1)                      ; Put-5
   (letrec ((lp1 (λ (d1a)
                   (d1a 5 (λ (d1b) (lp1 d1b))))))
     (lp1 d1)))
 (λ (u2 d2)                      ; Doubler
   (letrec ((lp2 (λ (u2a d2a)
                   (u2a (λ (x u2b)
                          (d2a (* 2 x)
                               (λ (d2b)
                                 (lp2 u2b d2b))))))))
     (lp2 u2 d2))))
```

(compose/pull put-5 doubler)

```
((λ (c1 c2)                            ; Compose/pull
   (λ (k u d) (c2 (λ (d') (c1 d')) d)))
 (λ (d1)                          ; Put-5
   (letrec ((lp1 (λ (d1a)
                   (d1a 5 (λ (d1b) (lp1 d1b))))))
     (lp1 d1)))
 (λ (u2 d2)                            ; Doubler
   (letrec ((lp2 (λ (u2a d2a)
                   (u2a (λ (x u2b)
                          (d2a (* 2 x)
                               (λ (d2b)
                                 (lp2 u2b d2b))))))))
     (lp2 u2 d2))))
```

η-reduce (1935)

(compose/pull put-5 doubler)

```
((λ (c1 c2)                        ; Compose/pull
   (λ (k u d) (c2 c1 d)))
 (λ (d1)                     ; Put-5
   (letrec ((lp1 (λ (d1a)
                   (d1a 5 lp1))))
     (lp1 d1)))
 (λ (u2 d2)                       ; Doubler
   (letrec ((lp2 (λ (u2a d2a)
                   (u2a (λ (x u2b)
                          (d2a (* 2 x)
                               (λ (d2b)
                                 (lp2 u2b d2b))))))))
     (lp2 u2 d2))))
```

(compose/pull put-5 doubler)

```
((λ (c1 c2)                          ; Compose/pull
   (λ (k u d) (c2 c1 d)))
 (λ (d1)                       ; Put-5
   (letrec ((lp1 (λ (d1a)
                   (d1a 5 lp1))))
     (lp1 d1)))
 (λ (u2 d2)                       ; Doubler
   (letrec ((lp2 (λ (u2a d2a)
                   (u2a (λ (x u2b)
                          (d2a (* 2 x)
                               (λ (d2b)
                                 (lp2 u2b d2b))))))))
     (lp2 u2 d2))))
```

β-reduce whole thing (1935)

(compose/pull put-5 doubler)

```
(λ (k u d)
  ((λ (u2 d2)                    ; Doubler
     (letrec ((lp2 (λ (u2a d2a)
                     (u2a (λ (x u2b)
                            (d2a (* 2 x)
                                 (λ (d2b)
                                   (lp2 u2b d2b))))))))
       (lp2 u2 d2)))
   (λ (d1)                     ; Put-5
     (letrec ((lp1 (λ (d1a) (d1a 5 lp1))))
       (lp1 d1)))
   d))
```

(compose/pull put-5 doubler)

```
(λ (k u d)
  ((λ (u2 d2)                         ; Doubler
     (letrec ((lp2 (λ (u2a d2a)
                     (u2a (λ (x u2b)
                            (d2a (* 2 x)
                                 (λ (d2b)
                                   (lp2 u2b d2b))))))))
       (lp2 u2 d2)))
   (λ (d1)                         ; Put-5
     (letrec ((lp1 (λ (d1a) (d1a 5 lp1))))
       (lp1 d1)))
   d))
```

β again (1935)

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b d2b))))))))
    (lp2 (λ (d1)                   ; Put-5
           (letrec ((lp1 (λ (d1a) (d1a 5 lp1))))
             (lp1 d1)))
         d)))
```

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b d2b)))))))))
    (lp2 (λ (d1)                       ; Put-5
           (letrec ((lp1 (λ (d1a) (d1a 5 lp1))))
             (lp1 d1)))
         d)))
```

Hoist inner `letrec`. (1980's)

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b d2b)))))))
           (lp1 (λ (d1a) (d1a 5 lp1))))
    (lp2 (λ (d1) (lp1 d1))
         d)))
```

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b d2b)))))))
           (lp1 (λ (d1a) (d1a 5 lp1))))
    (lp2 (λ (d1) (lp1 d1))
         d)))
```

η-reduce (1935)

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b d2b)))))))
           (lp1 (λ (d1a) (d1a 5 lp1))))
    (lp2 lp1
         d)))
```

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b d2b)))))))
           (lp1 (λ (d1a) (d1a 5 lp1))))
    (lp2 lp1
         d)))
```

Super-β: `u2a` = `u2b` = `lp1` (2006)

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (lp1 (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 lp1 d2b)))))))
           (lp1 (λ (d1a) (d1a 5 lp1))))
    (lp2 lp1 d)))
```

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (lp1 (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 lp1 d2b)))))))
           (lp1 (λ (d1a) (d1a 5 lp1))))
    (lp2 lp1 d)))
```

Eliminate useless `u2a`, `u2b`.

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (d2a)
                 (lp1 (λ (x)
                        (d2a (* 2 x)
                             (λ (d2b)
                               (lp2 d2b)))))))
           (lp1 (λ (d1a) (d1a 5))))
    (lp2 d)))
```

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (d2a)
                  (lp1 (λ (x)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 d2b)))))))
           (lp1 (λ (d1a) (d1a 5))))
    (lp2 d)))
```

η-reduce. (1935)

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (d2a)
                  (lp1 (λ (x)
                         (d2a (* 2 x) lp2)))))
           (lp1 (λ (d1a) (d1a 5))))
    (lp2 d)))
```

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (d2a)
                  (lp1 (λ (x)
                         (d2a (* 2 x) lp2)))))
           (lp1 (λ (d1a) (d1a 5))))
    (lp2 d)))
```

Inline & β-reduce `lp1` application. (1935)

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (d2a)
                  ((λ (d1a) (d1a 5))
                   (λ (x) (d2a (* 2 x) lp2))))))
    (lp2 d)))
```

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (d2a)
                 ((λ (d1a) (d1a 5))
                  (λ (x) (d2a (* 2 x) lp2))))))
    (lp2 d)))
```

Two more β steps. (1935)

(compose/pull put-5 doubler)

```
(λ (k u d)
  (letrec ((lp2 (λ (d2a)
                  (d2a (* 2 5) lp2))))
    (lp2 d)))
```

Liftoff!

Issues

- Linear “pipeline” topology wired in. Can we generalise?
- Can it be typed?
- OK, it works “by hand.” Can it be *implemented*?

Issues

- Linear “pipeline” topology wired in. Can we generalise?
- Can it be typed?
- OK, it works “by hand.” Can it be *implemented*?

Yes.

Channels in CPS

Explicit channels permit non-linear control/data-flow topologies.

Same optimisation story applies as in 3CPS case.

Types for functional coroutines

(α, β) `Channel` `/*` *coroutine connection: send an* α*, get a* β*.* `*/`

`switch` : $\alpha \times (\alpha, \beta)$ `Channel` $\rightarrow \beta \times (\alpha, \beta)$ `Channel`

```
datatype (α,β) Channel =
   Chan of (α * (β,α) Channel) cont;

fun switch(x, Chan k) =
   callcc (fn k' => throw k (x, Chan k'));
```

Details are in the paper.

Composing non-iterative computations

Some producers are truly recursive:

```
(define (gen-fringe tree chan)
  (if (leaf? tree)
      (put (leaf:val tree) chan)
      (let ((chan (gen-fringe (tree:left tree) chan)))
        (gen-fringe (tree:right tree) chan))))
```

What if we compose with summing consumer?

Composing non-iterative computations

Some producers are truly recursive:

```
(define (gen-fringe tree chan)
  (if (leaf? tree)
      (put (leaf:val tree) chan)
      (let ((chan (gen-fringe (tree:left tree) chan)))
        (gen-fringe (tree:right tree) chan))))
```

What if we compose with summing consumer?

Prototype compiler produces recursive, tree-walk summation.

Experience

- Built prototype compiler for toy dialect of Scheme.
  - Direct-style front end
  - Includes `call/cc`
  - Standard optimisations (β, η, ...)
  - Plus ΔCFA (POPL 2006), abstract GC, abstract counting (ΓCFA, ICFP 2006)
- Used for testing out Ph.D. analyses/optimisations
  Nothing transducer/coroutine specific—just a machine for attacking CPS.
- Successfully fuses put5/doubler, integrators, (rendered with coroutines/channels)
- Limiting reagent: Super-β.

Related work

Transducer fusion

- Deforestation
- Haskell's fold/build, unfold/destroy, *etc.*.
- Clu loop generators
- APL
- Filter fusion / Integrated layer processing

Final thoughts

- It's all about the representation.
  - λ as essential control/env/data-structure
  - CPS $\Rightarrow$ Our *main* concern
    becomes our *only* concern.

Once in CPS, generic optimisations suffice.

Final thoughts

- It's all about the representation.
  - λ as essential control/env/data-structure
  - CPS $\Rightarrow$ Our *main* concern
    becomes our *only* concern.

Once in CPS, generic optimisations suffice.
This generalises to exotic control structures.

Final thoughts

- It's all about the representation.
  - λ as essential control/env/data-structure
  - CPS $\Rightarrow$ Our *main* concern
becomes our *only* concern.

  Once in CPS, generic optimisations suffice.
  This generalises to exotic control structures.
- Coroutines are the neglected control structure.

Final thoughts

- It's all about the representation.
  - λ as essential control/env/data-structure
  - CPS $\Rightarrow$ Our *main* concern
    becomes our *only* concern.

  Once in CPS, generic optimisations suffice.
  This generalises to exotic control structures.
- Coroutines are the neglected control structure.
- Coroutines don't have to be heavyweight.
  (λ, CPS & static analysis are answer to efficiency issues.)

Final thoughts

- It's all about the representation.
  - λ as essential control/env/data-structure
  - CPS $\Rightarrow$ Our *main* concern
    becomes our *only* concern.

  Once in CPS, generic optimisations suffice.
  This generalises to exotic control structures.
- Coroutines are the neglected control structure.
- Coroutines don't have to be heavyweight.
  (λ, CPS & static analysis are answer to efficiency issues.)
- Lots to do! (Stay tuned)
  - Full-blown SML compiler
  - TCP/IP (Foxnet)
  - DSP libs.

Thank you.