Delimited Continuations in CS and Linguistics[1]

Oleg Kiselyov (FNMOC)
Chung-chieh Shan (Rutgers University)

December 4, 2007
Research Center for Language, Brain and Cognition
Tohoku University, Sendai, Japan

[1]Many helpful conversations with Rui Otake are gratefully acknowledged

?

Summary

Contexts and (delimited) control

Applications in Computer Science (backtracking, OS, Web,...)
Hints of linguistic applications

Dynamic Binding and Anaphora

Generating by jumping back-and-forth

Generating code, sentences, denotations in out-of-lexical-order

Type systems, CPS

CPS, double negation translation, type systems for ((delimited) control) effects formalize as a substructural logic
Types are abstract expressions (Cousot)
The colon is a turnstile (Lambek)

Code online

`http://okmij.org/ftp/Computation/Continuations.html`

Outline

- **Delimited continuations**

  Examining the stack

  Generating (sentences, meanings) by jumping back-and-forth

  CPS and types

  Summary

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print(42 + abs(2 * 3))
```

Full context	undelimited continuation function int $\rightarrow \infty$
Partial context	delimited continuation function int $\rightarrow$ int, i.e., take absolute value and add 42

Contexts and continuations are present whether we want them or not

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print(42 + abs(2 * 3))
```

Full context	undelimited continuation function int $\rightarrow \infty$
Partial context	delimited continuation function int $\rightarrow$ int, i.e., take absolute value and add 42

Contexts and continuations are present whether we want them or not

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print( 42 + abs( 2 * 3 ) )
```

Full context	undelimited continuation function int $\to \infty$
Partial context	delimited continuation function int $\to$ int, i.e., take absolute value and add 42

Contexts and continuations are present whether we want them or not

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print(42 + abs(6))
```

Contexts and continuations are present whether we want them or not

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print( 42 + if  6>0  then 6 else neg(6) )
```

Contexts and continuations are present whether we want them or not

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print( 42 + if true then 6 else neg(6) )
```

Contexts and continuations are present whether we want them or not

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print( 42 + 6 )
```

Contexts and continuations are present whether we want them or not

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

```
print( 48 )
```

Contexts and continuations are present whether we want them or not

Continuations are the meanings of evaluation contexts

A context is an expression with a hole

`print(48)`

Contexts and continuations are present whether we want them or not

Control effects: Process scheduling in OS

Operating system, User process, System call

```
schedule( main () {... read(file)  ...} ) ...
```

Control effects: Process scheduling in OS

Capture

```
schedule( main () {... read(file) ...} ) ...
```

```
schedule( ReadRequest(PCB ,file) ) ...
```

Control effects: Process scheduling in OS

Capture

```
schedule( main () {... read(file)  ...} ) ...
```

```
schedule( ReadRequest(PCB ,file) ) ...
```

...

```
schedule( resume(PCB ,"read string") ) ...
```

Control effects: Process scheduling in OS

Capture, Invoke

```
schedule( main () {... read(file) ...} ) ...
```

```
schedule( ReadRequest(PCB,file) ) ...
```

...

```
schedule( resume(PCB,"read string") ) ...
```

```
schedule( main () {... "read string" ...} ) ...
```

Control effects: Process scheduling in OS

Capture

```
schedule( main () {... read(file) ...} ) ...
```

```
schedule( ReadRequest( PCB ,file) ) ...
```

...

```
schedule( resume( PCB ,"read string") ) ...
```

```
schedule( main () {... "read string" ...} ) ...
```

User-level control operations ⇒ user-level scheduling, thread library

Control effects as debugging

```
debug_run( 42 + abs(2 * breakpt 1) )
```

Control effects as debugging

```
debug_run( 42 + abs(2 * breakpt 1) )
```

BP_1

Control effects as debugging

```
debug_run( 42 + abs(2 * breakpt 1 ) )
```

BP_1

```
debug_run(resume (BP1,3))
```

Control effects as debugging

```
debug_run( 42 + abs(2 * breakpt 1) )
```

BP_1

```
debug_run(resume (BP1,3))
```

```
debug_run( 42 + abs(2 * 3) )
```

first-class delimited continuations ⇒ a programmable debugger

- Back-tracking search (what if?), non-determinism
- Enumerator inversion: tracing a loop

Reset

“#” is the identity continuation (reset [ ]). “$” plugs in a term.

$$
\begin{aligned}
&\# \; \$ \text{ “Goldilocks said: ”} \frown \\
&\quad (\# \; \$ \text{ “This porridge is ”} \frown \text{“too hot”} \frown \text{“. ”}) \\
\rightsquigarrow\; &\# \; \$ \text{ “Goldilocks said: ”} \frown (\# \; \$ \text{ “This porridge is ”} \frown \text{“too hot. ”}) \\
\rightsquigarrow\; &\# \; \$ \text{ “Goldilocks said: ”} \frown (\# \; \$ \text{ “This porridge is too hot. ”}) \\
\rightsquigarrow\; &\# \; \$ \text{ “Goldilocks said: ”} \frown \text{“This porridge is too hot. ”} \\
\rightsquigarrow\; &\# \; \$ \text{ “Goldilocks said: This porridge is too hot. ”} \\
\rightsquigarrow\; &\text{“Goldilocks said: This porridge is too hot. ”}
\end{aligned}
$$

Shift

"出k." removes and binds k to a continuation.

$$\# \$ \text{“Goldilocks said: ”} \frown \\ (\# \$ \text{“This porridge is ”} \frown \\ (\text{出}k.(k \$ \text{“too hot”}) \frown (k \$ \text{“too cold”}) \frown (k \$ \text{“just right”})) \\ \frown \text{“. ”})$$

$$\leadsto \# \$ \text{“Goldilocks said: ”} \frown \\ (\# \$ ((\# \$ \text{“This porridge is ”} \frown [\,] \frown \text{“. ”}) \$ \text{“too hot”}) \frown \\ ((\# \$ \text{“This porridge is ”} \frown [\,] \frown \text{“. ”}) \$ \text{“too cold”}) \frown \\ ((\# \$ \text{“This porridge is ”} \frown [\,] \frown \text{“. ”}) \$ \text{“just right”}))$$

Shift

"出k." removes and binds k to a continuation.

$$\# \; \$ \; \text{“Goldilocks said: ”} \frown$$
$$(\# \; \$ \; \text{“This porridge is ”} \frown$$
$$(出k.(k \; \$ \; \text{“too hot”}) \frown (k \; \$ \; \text{“too cold”}) \frown (k \; \$ \; \text{“just right”}))$$
$$\frown \text{“. ”})$$

$$\rightsquigarrow \# \; \$ \; \text{“Goldilocks said: ”} \frown$$
$$(\# \; \$ \; (\# \; \$ \; \text{“This porridge is ”} \frown \text{“too hot”} \frown \text{“. ”}) \frown$$
$$(\# \; \$ \; \text{“This porridge is ”} \frown \text{“too cold”} \frown \text{“. ”}) \frown$$
$$(\# \; \$ \; \text{“This porridge is ”} \frown \text{“just right”} \frown \text{“. ”}))$$

$$\rightsquigarrow \cdots$$

$\rightsquigarrow$ "Goldilocks said:
This porridge is too hot.
This porridge is too cold.
This porridge is just right. "

Terms	$E, F ::= V \mid FE \mid C \,\$\, E \mid 出k.\, E$
Values	$V ::= x \mid \lambda x.\, E$
Coterms	$C ::= k \mid \# \mid E, C \mid C; V$
Types	$T ::= U \mid S \downarrow T$
Pure types	$U ::= U \to T \mid \text{string} \mid \text{int} \mid \cdots$
Cotypes	$S ::= U \uparrow T$

Transitions

$$C_1 \,\$\, \cdots \,\$\, C_n \,\$\, (\lambda x.\, E)V \rightsquigarrow C_1 \,\$\, \cdots \,\$\, C_n \,\$\, E\{x \mapsto V\}$$

$$C_1 \,\$\, \cdots \,\$\, C_n \,\$\, C \,\$\, (出k.\, E) \rightsquigarrow C_1 \,\$\, \cdots \,\$\, C_n \,\$\, \# \,\$\, E\{k \mapsto C\}$$

Structural rules express evaluation order

$$C \,\$\, FE = E, C \,\$\, F \qquad C \,\$\, VE = C; V \,\$\, E \qquad V = \# \,\$\, V$$

$$\begin{aligned} \# \,\$\, \big(V_1(V_2V_3)\big)V_4 &= (V_4, \#) \,\$\, V_1(V_2V_3) \\ &= \big(V_2V_3, (V_4, \#)\big) \,\$\, V_1 \\ &= \big((V_4, \#); V_1\big) \,\$\, V_2V_3 \end{aligned}$$

Our coterm type $T \uparrow T'$ is $T' /_{\$} T$.

Our impure term type $T \downarrow T'$ is $T \backslash_{\$} T'$.

Reset: dynamic semantics

Alternate between refocusing and reducing.

$$
\begin{aligned}
& \#\,\$\ \text{“Goldilocks said: ”} \frown \\
& \quad (\#\,\$\ \text{“This porridge is ”} \frown \text{“too hot”} \frown \text{“. ”}) \\
= {} & \#;(\text{“Goldilocks said: ”}\frown)\ \$ \\
& \quad (\#;(\text{“This porridge is ”}\frown)\ \$\ \text{“too hot”} \frown \text{“. ”}) \\
\leadsto {} & \#;(\text{“Goldilocks said: ”}\frown)\ \$ \\
& \quad (\#;(\text{“This porridge is ”}\frown)\ \$\ \text{“too hot. ”}) \\
= {} & \#;(\text{“Goldilocks said: ”}\frown)\ \$\ (\#\,\$\ \text{“This porridge is ”} \frown \text{“too hot. ”}) \\
\leadsto {} & \#;(\text{“Goldilocks said: ”}\frown)\ \$\ (\#\,\$\ \text{“This porridge is too hot. ”}) \\
= {} & \#\,\$\ \text{“Goldilocks said: ”} \frown \text{“This porridge is too hot. ”} \\
\leadsto {} & \#\,\$\ \text{“Goldilocks said: This porridge is too hot. ”} \\
= {} & \text{“Goldilocks said: This porridge is too hot. ”}
\end{aligned}
$$

Shift: dynamic semantics

$$
\begin{aligned}
&\#\ \$\ \text{“Goldilocks said: ”} \frown \\
&\quad (\#\ \$\ \text{“This porridge is ”} \frown \\
&\qquad (\text{出}k.(k\ \$\ \text{“too hot”}) \frown (k\ \$\ \text{“too cold”}) \frown (k\ \$\ \text{“just right”})) \\
&\hfill \frown \text{“. ”})
\end{aligned}
$$

$$
\begin{aligned}
= {} & \#; (\text{“Goldilocks said: ”} \frown)\ \$ \\
& \quad ((\text{“. ”}, (\#; (\text{“This porridge is ”} \frown))); \frown)\ \$ \\
& \qquad (\text{出}k.(k\ \$\ \text{“too hot”}) \frown (k\ \$\ \text{“too cold”}) \frown (k\ \$\ \text{“just right”}))
\end{aligned}
$$

$$
\begin{aligned}
\rightsquigarrow {} & \#; (\text{“Goldilocks said: ”} \frown)\ \$\ \#\ \$ \\
& \quad ((((\text{“. ”}, (\#; (\text{“This porridge is ”} \frown))); \frown)\ \$\ \text{“too hot”}) \frown \\
& \quad\ (((\text{“. ”}, (\#; (\text{“This porridge is ”} \frown))); \frown)\ \$\ \text{“too cold”}) \frown \\
& \quad\ (((\text{“. ”}, (\#; (\text{“This porridge is ”} \frown))); \frown)\ \$\ \text{“just right”}))
\end{aligned}
$$

$$
\begin{aligned}
= {} & \#; (\text{“Goldilocks said: ”} \frown)\ \$\ \#\ \$ \\
& \quad ((\#\ \$\ \text{“This porridge is ”} \frown \text{“too hot”} \frown \text{“. ”}) \frown \\
& \quad\ (\#\ \$\ \text{“This porridge is ”} \frown \text{“too cold”} \frown \text{“. ”}) \frown \\
& \quad\ (\#\ \$\ \text{“This porridge is ”} \frown \text{“just right”} \frown \text{“. ”}))
\end{aligned}
$$

$\rightsquigarrow \cdots$

Outline

Delimited continuations

▶ **Examining the stack**

Generating (sentences, meanings) by jumping back-and-forth

CPS and types

Summary

Dynamic binding: summary

Many applications

- Implicit arguments: the-*current*-directory, *thepage*
- I/O redirection
- Exception handlers
- Mobile code
- Web applications
- Linguistics: the topic, anaphora
- …

Dynamic binding: summary

Many applications

- Implicit arguments: the-*current*-directory, *thepage*
- I/O redirection
- Exception handlers
- Mobile code
- Web applications
- Linguistics: the topic, anaphora
- …

Many implementations

- Pass implicit argument (*dynamic environment*) everywhere
- Global mutable cells (*shallow binding*)
- …

Dynamic binding: summary

Many applications

- Implicit arguments: the-*current*-directory, *thepage*
- I/O redirection
- Exception handlers
- Mobile code
- Web applications
- Linguistics: the topic, anaphora
- …

Many implementations

- Pass implicit argument (*dynamic environment*) everywhere
- Global mutable cells (*shallow binding*)
- …

Dynamic binding: summary

Many applications

- Implicit arguments: the-*current*-directory, *thepage*
- I/O redirection
- Exception handlers
- Mobile code
- Web applications
- Linguistics: the topic, anaphora
- …

Many implementations

- Pass implicit argument (*dynamic environment*) everywhere
- Global mutable cells (*shallow binding*)
- …

context as an implicit, ever-present argument

Anaphora and context marks

Goldilocks said the porridge is too hot for her.

Anaphora and context marks

"Goldilocks" $\frown$ " said the porridge is too hot."

Anaphora and context marks

$(\text{“Goldilocks”}^\frown)(\#\$\text{“ said the porridge is too hot.”})$

$\leadsto$ “Goldilocks said the porridge is too hot.”

Anaphora and context marks

(interp “Goldilocks”)(# $ String “ said the porridge is too hot.”)

```
interp str = function
  | String x -> str ⌒ x
```

Anaphora and context marks

(interp "Goldilocks")
(#$" said the porridge is too hot " $\frown$ "for " $\frown$ *her* $\frown$ ".")

```
interp str = function
  | String x -> str ⌢ x
```

Anaphora and context marks

(interp "Goldilocks")
  (#$ " said the porridge is too hot " $\frown$ "for " $\frown$
    (出k. Req(Female, k)) $\frown$ ".")

```
interp str = function
  | String x -> str ⌒ x
  | Req(Female,k) -> interp str (k $ str)
```

Anaphora and context marks

(interp "Goldilocks")
$(\#\$$" said the porridge is too hot " $\frown$ "for " $\frown$
$(出 k.\, \mathrm{Req}(\mathrm{Female}, k))$ $\frown$ ".")

$\rightsquigarrow$

(interp "Goldilocks")$(\#\ \$\ \mathrm{Req}(\mathrm{Female}, k))$

```
interp str = function
  | String x -> str ⌢ x
  | Req(Female,k) -> interp str (k $ str)
```

Anaphora and context marks

(interp "Goldilocks")
  (#$" said the porridge is too hot " ⌢ "for " ⌢
    (出k. Req(Female, k)) ⌢ ".")

$\rightsquigarrow$

(interp "Goldilocks")(# $ Req(Female, k))

$\rightsquigarrow$

(interp "Goldilocks")
(# $ " said the porridge is too hot " ⌢ "for " ⌢ "Goldilocks" ⌢ ".")

```
interp str = function
  | String x -> str ⌢ x
  | Req(Female,k) -> interp str (k $ str)
```

Anaphora and context marks

(interp "Goldilocks")
 (#$" said the porridge is too hot " $\frown$ "for " $\frown$
 (出k. Req(Female, k)) $\frown$ ".")

$\rightsquigarrow$

(interp "Goldilocks")(# $ Req(Female, k))

$\rightsquigarrow$

(interp "Goldilocks")
(# $ " said the porridge is too hot " $\frown$ "for " $\frown$ "Goldilocks" $\frown$ ".")

$\rightsquigarrow$ "Goldilocks said the porridge is too hot for Goldilocks."

```
interp str = function
  | String x -> str ⌢ x
  | Req(Female,k) -> interp str (k $ str)
```

Several Pronouns, Several Marks

Goldilocks tasted the porridge and said that it is too hot for her.

Several Pronouns, Several Marks

Goldilocks tasted the porridge and said that it is too hot for her.

(interp Female "Goldilocks")
  (# $ " tasted " ⌢ ((interp Thing "the porridge")
    (# $ " and said that " ⌢ (出k. Req(Thing, k))⌢
      " is too hot for " ⌢ (出k. Req(Female, k)) ⌢ ".")))

```
interp mytag str = function
  | String x -> str ^ x
  | Req(tag,k) when tag = mytag ->
     interp mytag str (k $ str)
```

Several Pronouns, Several Marks

$$(\text{interp}\,\text{Female}\,\text{“Goldilocks”})$$
$$(\#\,\$\,\text{“ tasted ”} \frown ((\text{interp}\,\text{Thing}\,\text{“the porridge”})$$
$$(\#\,\$\,\text{“ and said that ”} \frown (出k.\,\text{Req}(\text{Thing}, k)) \frown$$
$$\text{“ is too hot for ”} \frown (出k.\,\text{Req}(\text{Female}, k)) \frown \text{“.”})))$$

$\rightsquigarrow$

$$(\text{interp}\,\text{Female}\,\text{“Goldilocks”})$$
$$(\#\,\$\,\text{“ tasted ”} \frown ((\text{interp}\,\text{Thing}\,\text{“the porridge”})$$
$$(\#\,\$\,\text{Req}(\text{Thing}, k_1))))$$

```
interp mytag str = function
  | String x -> str ⌢ x
  | Req(tag,k) when tag = mytag ->
    interp mytag str (k $ str)
```

Several Pronouns, Several Marks

$\rightsquigarrow$

(interp Female "Goldilocks")
  (# $ " tasted " ⌢ ((interp Thing "the porridge")
    (# $ " and said that " ⌢ "the porridge" ⌢
      " is too hot for " ⌢ (出k. Req(Female, k)) ⌢ ".")))

```
interp mytag str = function
  | String x -> str ⌢ x
  | Req(tag,k) when tag = mytag ->
     interp mytag str (k $ str)
```

Several Pronouns, Several Marks

$\rightsquigarrow$

(interp Female "Goldilocks")
  (# \$ " tasted " ⌢ ((interp Thing "the porridge")
    (# \$ " and said that the porridge is too hot for " ⌢
      (出k. Req(Female, k)) ⌢ ".")))

```
interp mytag str = function
  | String x -> str ^ x
  | Req(tag,k) when tag = mytag ->
     interp mytag str (k $ str)
```

Several Pronouns, Several Marks

$\leadsto$

(interp Female "Goldilocks")
  (# \$ " tasted " $\frown$ ((interp Thing "the porridge")
    (# \$ " and said that the porridge is too hot for " $\frown$
      (出k. Req(Female, k)) $\frown$ ".")))

$\leadsto$

(interp Female "Goldilocks")
  (# \$ " tasted " $\frown$ ((interp Thing "the porridge")
    (# \$ Req(Female, k_2))))

```
interp mytag str = function
  | String x -> str ⌢ x
  | Req(tag,k) when tag = mytag ->
    interp mytag str (k $ str)
  | Req(tag,k) ->
    let v = 出k. Req(tag, k) in interp mytag str (k $ v)
```

Several Pronouns, Several Marks

$\leadsto$

(interp Female "Goldilocks")
  (# \$ " tasted " $\frown$
    (*let* v = 出k. Req(Female, k) *in*
    interp Thing "the porridge" (k_2 \$ v)))

```
interp mytag str = function
  | String x -> str ⌒ x
  | Req(tag,k) when tag = mytag ->
     interp mytag str (k $ str)
  | Req(tag,k) ->
     let v = 出k. Req(tag, k) in interp mytag str (k $ v)
```

Several Pronouns, Several Marks

$\rightsquigarrow$

(interp Female "Goldilocks")
(# \$ " tasted " ⌢
(*let* $v = 出k.\,\mathrm{Req}(\mathrm{Female}, k)$ *in*
interp Thing "the porridge" (k_2 \$ v)))

$\rightsquigarrow$

(interp Female "Goldilocks")
(# \$ Req(Female, k_3))

```
interp mytag str = function
  | String x -> str ⌢ x
  | Req(tag,k) when tag = mytag ->
     interp mytag str (k $ str)
  | Req(tag,k) ->
     let v = 出k. Req(tag, k) in interp mytag str (k $ v)
```

Several Pronouns, Several Marks

$\leadsto$

(interp Female "Goldilocks")
  (# \$ " tasted " $\frown$
    (*let* v = "Goldilocks" *in*
    interp Thing "the porridge" (k_2 \$ v)))

```
interp mytag str = function
  | String x -> str ⌢ x
  | Req(tag,k) when tag = mytag ->
     interp mytag str (k $ str)
  | Req(tag,k) ->
     let v = 出k. Req(tag, k) in interp mytag str (k $ v)
```

Several Pronouns, Several Marks

$\rightsquigarrow$

(interp Female "Goldilocks")
  (# \$ " tasted " ⌢ ((interp Thing "the porridge")
    (# \$ " and said that the porridge is too hot for " ⌢
      "Goldilocks" ⌢ ".")))

$\rightsquigarrow$

```
interp mytag str = function
  | String x -> str ⌢ x
  | Req(tag,k) when tag = mytag ->
    interp mytag str (k $ str)
  | Req(tag,k) ->
    let v = 出k. Req(tag, k) in interp mytag str (k $ v)
```

Several Pronouns, Several Marks

$\rightsquigarrow$

(interp Female “Goldilocks”)
 (# $ “ tasted ” ⌢ ((interp Thing “the porridge”)
  (# $ “ and said that the porridge is too hot for ” ⌢
   “Goldilocks” ⌢ “.”)))

$\rightsquigarrow$

Goldilocks tasted the porridge and said that the porridge is too hot for Goldilocks.

```
interp mytag str = function
  | String x -> str ⌢ x
  | Req(tag,k) when tag = mytag ->
     interp mytag str (k $ str)
  | Req(tag,k) ->
     let v = 出k. Req(tag, k) in interp mytag str (k $ v)
```

Far-reaching pronouns

need to look past the immediate occurrence

“he gave this to him”

Far-reaching pronouns

need to look past the immediate occurrence

“Now just one thing more remained, the box that held the daylight, and he cried for that. His eyes turned around and showed different colors, and the people began thinking that he must be something other than an ordinary baby. But it always happens that a grandfather loves his grandchild just as he does his own daughter, so the grandfather felt very sad when he gave this to him. When the child had this in his hands, he uttered the raven cry, ”Ga,” and flew out with it through the smoke hole.”

“Raven”, Tlingit Indians of Southeastern Alaska

Far-reaching pronouns

```
interp mytag str = function
  | String x -> str ^ x
  | Req(tag,k) when tag = mytag ->
     interp mytag str (k $ str)
  | Req(tag,k) ->
     let v = 出k. Req(tag,k) in interp mytag str (k $ v)
  | ReqDefer(fn,k) ->
     let v = fn str in interp mytag str (k $ v)
```

Leaving bread-crumbs on the stack, walking the stack and examining them

Anaphora and *dynamic* binding

Aspects of dynamism:

1. Examining any number of previous bindings
2. Referring to a binding occurrence that is not in scope (e.g., referring to a noun in a clause)
   Solution: “binding that moves itself up”, see next

Outline

- Delimited continuations
- Examining the stack
- **Generating (sentences, meanings) by jumping back-and-forth**
- CPS and types
- Summary

Generating denotations of questions

これは · (本 · です)
$\rightsquigarrow$ *this* · (*is*($\lambda e.\, e$ · a-book))

```
let (·) x f = f x
let make_app x f = x ⌢ ⌜·⌝ ⌢ f
let これは = ⌜this⌝
let 本 e = make_app e ⌜a-book⌝
let です f = fun e -> make_app e ⌜(is(λe.⌝ ⌢ (f ⌜e⌝) ⌢ ⌜))⌝
let だ f = fun e -> make_app e ⌜(is(λe.⌝ ⌢ (f ⌜e⌝) ⌢ ⌜))⌝
```

```
this   : e
a-book : et
is     : (et)(et)
```

Generating denotations of questions

(これは · (何 · です)) · か

```
let (·) x f = f x
let make_app x f = x ⌢ ⌜.⌝ ⌢ f
let これは = ⌜this⌝
let 本 e = make_app e ⌜a-book⌝
let です f = fun e -> make_app e ⌜(is(λe.⌝ ⌢ (f ⌜e⌝) ⌢ ⌜))⌝
let だ f = fun e -> make_app e ⌜(is(λe.⌝ ⌢ (f ⌜e⌝) ⌢ ⌜))⌝
```

```
this   : e
a-book : et
is     : (et)(et)
```

Generating denotations of questions

(これは · (何 · です)) · か
$\rightsquigarrow (\lambda x.\, this \cdot (is(\lambda e.\, e \cdot x)))$

let (·) x f = f x
let make_app x f = $x \frown \ulcorner \cdot \urcorner \frown f$
let これは = $\ulcorner this \urcorner$
let 本 e = make_app e $\ulcorner$a-book$\urcorner$
let です f = fun e -> make_app e $\ulcorner (is(\lambda e. \urcorner \frown (f\ \ulcorner e \urcorner) \frown \ulcorner)) \urcorner$
let だ f = fun e -> make_app e $\ulcorner (is(\lambda e. \urcorner \frown (f\ \ulcorner e \urcorner) \frown \ulcorner)) \urcorner$
let 何 = 出$k. \ulcorner (\lambda x. \urcorner \frown (k \,\$\, (\lambda e.\, make_app\ \ e \ulcorner x \urcorner)) \frown \ulcorner) \urcorner)$
let か f = $\#\, \$\, f$

```
this   : e
a-book : et
is     : (et)(et)
```

Generating denotations of questions

(これは · (本 · だ)) · と言いました
⇝ (*this* · (*is*(λ*e*. *e* · a-book))) · so-he-said

```
let (·) x f = f x
let make_app x f = x ⁀ ⌜·⌝ ⁀ f
let これは = ⌜this⌝
let 本 e = make_app e ⌜a-book⌝
let です f = fun e -> make_app e ⌜(is(λe.⌝ ⁀ (f ⌜e⌝) ⁀ ⌜))⌝
let だ f = fun e -> make_app e ⌜(is(λe.⌝ ⁀ (f ⌜e⌝) ⁀ ⌜))⌝
let 何 = 出k.⌜(λx.⌝ ⁀ (k $ (λe. make_app e⌜x⌝)) ⁀ ⌜)⌝)
let か f = # $ f
let と言いました f = make_app (⌜(⌝ ⁀ f() ⁀ ⌜)⌝) ⌜so-he-said⌝
```

```
this   : e
a-book : et
is     : (et)(et)
```

Generating denotations of questions

((これは · (何 · だ)) · と言いました) · か

```
let (·) x f = f x
let make_app x f = x ⁀ ⌜·⌝ ⁀ f
let これは = ⌜this⌝
let 本 e = make_app e ⌜a-book⌝
let です f = fun e -> make_app e ⌜(is(λe.⌝ ⁀ (f ⌜e⌝) ⁀ ⌜))⌝
let だ f = fun e -> make_app e ⌜(is(λe.⌝ ⁀ (f ⌜e⌝) ⁀ ⌜))⌝
let 何 = 出k.⌜(λx.⌝ ⁀ (k $ (λe. make_app e⌜x⌝)) ⁀ ⌜)⌝)
let か f = # $ f
let と言いました f = make_app (⌜(⌝ ⁀ f() ⁀ ⌜)⌝) ⌜so-he-said⌝
```

Generating denotations of questions

((これは · (何 · だ)) · と言いました) · か
$\rightsquigarrow (\lambda x.(\mathit{this} \cdot (\mathit{is}(\lambda e.\, e \cdot x))) \cdot \text{so-he-said})$

```
let (·) x f = f x
let make_app x f = x ⌢ ⌜·⌝ ⌢ f
let これは = ⌜this⌝
let 本 e = make_app e ⌜a-book⌝
let です f = fun e -> make_app e ⌜(is(λe.⌝ ⌢ (f ⌜e⌝) ⌢ ⌜))⌝
let だ f = fun e -> make_app e ⌜(is(λe.⌝ ⌢ (f ⌜e⌝) ⌢ ⌜))⌝
let 何 = 出k.⌜(λx.⌝ ⌢ (k $ (λe. make_app e⌜x⌝)) ⌢ ⌜)⌝)
let か f = # $ f
let と言いました f = make_app (⌜(⌝ ⌢ f() ⌢ ⌜)⌝) ⌜so-he-said⌝
```

Generating denotations of questions

((これは · (何 · だ)) · と言いました) · か
$\rightsquigarrow (\lambda x.(\mathit{this} \cdot (\mathit{is}(\lambda e.\, e \cdot x))) \cdot \text{so-he-said})$
(((これは · (何 · です)) · か) · と言いました)

```
let (·) x f = f x
let make_app x f = x ⌢ ⌜·⌝ ⌢ f
let これは = ⌜this⌝
let 本 e = make_app e ⌜a-book⌝
let です f = fun e -> make_app e ⌜(is(λe.⌝ ⌢ (f ⌜e⌝) ⌢ ⌜))⌝
let だ f = fun e -> make_app e ⌜(is(λe.⌝ ⌢ (f ⌜e⌝) ⌢ ⌜))⌝
let 何 = 出k.⌜(λx.⌝ ⌢ (k $ (λe. make_app e⌜x⌝)) ⌢ ⌜)⌝)
let か f = # $ f
let と言いました f = make_app (⌜(⌝ ⌢ f() ⌢ ⌜)⌝) ⌜so-he-said⌝
```

Generating denotations of questions

((これは · (何 · だ)) · と言いました) · か
$\rightsquigarrow$ ($\lambda x.(this \cdot (is(\lambda e.\, e \cdot x)))$) · so-he-said)
(((これは · (何 · です)) · か) · と言いました)
$\rightsquigarrow$ ($\lambda x.(this \cdot (is(\lambda e.\, e \cdot x)))$) · so-he-said

```
let (·) x f = f x
let make_app x f = x ⌢ ⌜·⌝ ⌢ f
let これは = ⌜this⌝
let 本 e = make_app e ⌜a-book⌝
let です f = fun e -> make_app e ⌜(is(λe.⌝ ⌢ (f ⌜e⌝) ⌢ ⌜))⌝
let だ f = fun e -> make_app e ⌜(is(λe.⌝ ⌢ (f ⌜e⌝) ⌢ ⌜))⌝
let 何 = 出k.⌜(λx.⌝ ⌢ (k $ (λe. make_app e⌜x⌝)) ⌢ ⌜)⌝)
let か f = # $ f
let と言いました f = make_app (⌜(⌝ ⌢ f() ⌢ ⌜)⌝) ⌜so-he-said⌝
```

Outline

Delimited continuations

Examining the stack

Generating (sentences, meanings) by jumping back-and-forth

▶ **CPS and types**

Summary

Introduction to CPS

$42 < (2 \times \mathit{breakpt})$

The type of 42:

- int
- $(\text{int} \to \text{bool}) \to \text{bool}$
- $(\text{int} \to \alpha) \to \alpha$: context independence
- $(\text{int} \to F) \to F$

CPS and Double Negation

Glivenko's Theorem [1929]: An arbitrary propositional formula A is classically provable, if and only if $\neg\neg A$ is intuitionistically provable.

Answer types in the CPS transformation

$$1 < 2$$

$$\lambda k. \quad (\lambda k.\, k1) \quad (\lambda x. \quad (\lambda k.\, k2) \quad (\lambda y. \quad k(x < y) \quad))$$

Answer types in the CPS transformation

$$1 < 2$$

$$\overbrace{\lambda k.\ \underbrace{\overbrace{(\lambda k.\,k1)}^{(\mathrm{int}\to T)\to T}\ (\lambda x.\ \underbrace{\overbrace{(\lambda k.\,k2)}^{(\mathrm{int}\to T)\to T}\ (\lambda y.\ \underbrace{k(x<y)}_{T}\ ))}_{T}}_{T}}^{(\mathrm{bool}\to T)\to T}$$

Answer types in the CPS transformation

$$1 < 2$$

$$\overbrace{\lambda k.\ \underbrace{\overbrace{(\lambda k.\, k1)}^{(\mathrm{int} \to T) \to T}\ (\lambda x.\ \underbrace{\overbrace{(\lambda k.\, k2)}^{(\mathrm{int} \to T) \to T}\ (\lambda y.\ \underbrace{k(x < y)}_{T}\ ))}_{T}}_{T}}^{(\mathrm{bool} \to T) \to T}$$

Answer types in the CPS transformation

$$1 < 2$$

$$(\text{出}k.\ \text{“Ouch!”}) < 2$$

$$\overbrace{\lambda k.\ \underbrace{\overbrace{(\lambda k.\ \text{“Ouch!”})}^{(\text{int}\to T)\to\text{string}}\ (\lambda x.\ \underbrace{\overbrace{(\lambda k.\,k2)}^{(\text{int}\to T)\to T}\ (\lambda y.\ \underbrace{k(x<y)}_{T}\ ))}_{T}}_{\text{string}}}^{(\text{bool}\to T)\to\text{string}}$$

Answer types in the CPS transformation

$$1 < 2$$

$$(\text{出}k.\ \text{“Ouch!”}) < 2$$

$$1 < (\text{出}k.\ \text{‘c’})$$

$$\overbrace{\lambda k.\ \underbrace{\overbrace{(\lambda k.\, k1)}^{(\text{int}\to\text{char})\to\text{char}}\ (\lambda x.\ \underbrace{\overbrace{(\lambda k.\ \text{‘c’})}^{(\text{int}\to T)\to\text{char}}\ (\lambda y.\ \underbrace{k(x<y)}_{T}))}_{\text{char}}}_{\text{char}}}^{(\text{bool}\to T)\to\text{char}}$$

Answer types in the CPS transformation

$$1 < 2$$

$$(\text{出}k.\ \text{“Ouch!”}) < 2$$

$$1 < (\text{出}k.\ \text{‘c’})$$

$$(\text{出}k.\ \text{“Ouch!”}) < (\text{出}k.\ \text{‘c’})$$

$$\overbrace{\lambda k.\ \underbrace{\overbrace{(\lambda k.\ \text{“Ouch!”})}^{(\mathrm{int}\to\mathrm{char})\to\mathrm{string}}\ (\lambda x.\ \underbrace{\overbrace{(\lambda k.\ \text{‘c’})}^{(\mathrm{int}\to T)\to\mathrm{char}}\ (\lambda y.\ \underbrace{k(x<y)}_{T}\ ))}_{\mathrm{char}}}_{\mathrm{string}}}^{(\mathrm{bool}\to T)\to\mathrm{string}}$$

Evaluation order chains together *initial* and *final* answer types.

Outline

Delimited continuations

Examining the stack

Generating (sentences, meanings) by jumping back-and-forth

CPS and types

▶ **Summary**

Summary

Contexts and (delimited) control

Applications in Computer Science (backtracking, OS, Web,...)
Hints of linguistic applications

Dynamic Binding and Anaphora

Generating by jumping back-and-forth

Generating code, sentences, denotations in out-of-lexical-order

Type systems, CPS

CPS, double negation translation, type systems for ((delimited) control) effects formalize as a substructural logic
Types are abstract expressions (Cousot)
The colon is a turnstile (Lambek)

Code online

`http://okmij.org/ftp/Computation/Continuations.html`