Delimited Continuations in CS and Linguistics

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

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

1Many 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

\[
\text{print}( 42 + \text{abs}(2 \times 3) )
\]

**Full context**
- undelimited continuation function
- \( \text{int} \rightarrow \infty \)

**Partial context**
- delimited continuation function
- \( \text{int} \rightarrow \text{int} \), 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

\[
\text{print}(42 + \text{abs}(2 * 3))
\]

**Full context**
- undelimited continuation function
- \(\text{int} \rightarrow \infty\)

**Partial context**
- delimited continuation function
- \(\text{int} \rightarrow \text{int}\), 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

\[
\text{print}(42 + \text{abs}(2 \times 3))
\]

<table>
<thead>
<tr>
<th>Context Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full context</strong></td>
<td>undelimited continuation function</td>
</tr>
<tr>
<td></td>
<td>(\text{int} \rightarrow \infty)</td>
</tr>
<tr>
<td><strong>Partial context</strong></td>
<td>delimited continuation function</td>
</tr>
<tr>
<td></td>
<td>(\text{int} \rightarrow \text{int}), i.e., take absolute value and add 42</td>
</tr>
</tbody>
</table>

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

\[
\text{print}(42 + \text{if } 6 > 0 \text{ then } 6 \text{ else } \text{neg}(6))
\]

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

\[
\text{print}(42 + \text{if true then } 6 \text{ else } \text{neg}(6))
\]

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

\text{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

\[
\text{schedule( main () \{... read(file) ...\} ) ...}
\]

\[
\text{schedule( ReadRequest( PCB , file) ) ...}
\]
Control effects: Process scheduling in OS

Capture

```c
schedule( main () {... read(file) ...} ) ...
```
```c
schedule( ReadRequest( PCB , file ) ) ...
```
```c
... schedule( resume( PCB , "read string") ) ...
```
Control effects: Process scheduling in OS

Capture, Invoke

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

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

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

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

```plaintext
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

\[
\text{debug\_run}(42 + \text{abs}(2 * \text{breakpt}\ 1))
\]

BP\(_1\)
Control effects as debugging

\[
\text{debug\_run}(42 + \text{abs}(2 \times \text{breakpt} 1))
\]

\[BP_1\]

\[
\text{debug\_run}(\text{resume} \ (BP_1,3))
\]
Control effects as debugging

```python
def debug_run(expr):
    # This is a debug run function
    # with a breakpoint at 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
“#” is the identity continuation (reset [ ]). “$” plugs in a term.

# $ “Goldilocks said: ” \sim

\sim # $ “This porridge is ” \sim “too hot” \sim “. ”

\sim # $ “Goldilocks said: ” \sim ( # $ “This porridge is ” \sim “too hot. ”)

\sim # $ “Goldilocks said: ” \sim ( # $ “This porridge is too hot. ”)

\sim # $ “Goldilocks said: ” \sim “This porridge is too hot. ”

\sim # $ “Goldilocks said: This porridge is too hot. ”

\sim “Goldilocks said: This porridge is too hot. ”
“出$k.$” removes and binds $k$ to a continuation.

```
# $ "Goldilocks said: "  \(\sim\)
  ( # $ "This porridge is "  \(\sim\)
    (出$k.$(k $ "too hot")  \(\sim\) (k $ "too cold")  \(\sim\) (k $ "just right")))
  \(\sim\ "\)"
```

```
\(\sim\) # $ "Goldilocks said: "  \(\sim\)
  (# $ ((# $ "This porridge is "  \(\sim\) [ ]  \(\sim\) ".")) $ "too hot")  \(\sim\)
    ((# $ "This porridge is "  \(\sim\) [ ]  \(\sim\) ".")) $ "too cold")  \(\sim\)
    ((# $ "This porridge is "  \(\sim\) [ ]  \(\sim\) ".")) $ "just right")))
```
“出$k.$” removes and binds $k$ to a continuation.

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

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

\[
\leadsto \ldots
\]

\[
\leadsto \text{“Goldilocks said:}
\li \text{This porridge is too hot.}
\li \text{This porridge is too cold.}
\li \text{This porridge is just right. ”}
\]
Terms

\[ E, F ::= V | FE | C \uparrow E | \text{out } k. E \]

Values

\[ V ::= x | \lambda x. E \]

Coterms

\[ C ::= k | \# | E, C | C; V \]

Types

\[ T ::= U | S \downarrow T \]

Pure types

\[ U ::= U \rightarrow T | \text{string} | \text{int} | \cdots \]

Cotypes

\[ S ::= U \uparrow T \]

Transitions

\[
C_1 \cdots C_n \ (\lambda x. E)V \leadsto C_1 \cdots C_n \ E\{x \mapsto V\} \\
C_1 \cdots C_n \ C \ (\text{out } k. E) \leadsto C_1 \cdots C_n \ \# \ E\{k \mapsto C\}
\]
Structural rules express evaluation order

\[ C \downarrow FE = E, \ C \downarrow F \quad C \downarrow VE = C; V \downarrow E \quad V = \# \downarrow V \]

\[
\# \downarrow (V_1(V_2V_3))V_4 = (V_4, \#) \downarrow V_1(V_2V_3) \\
= (V_2V_3, (V_4, \#)) \downarrow V_1 \\
= ((V_4, \#); V_1) \downarrow V_2V_3
\]

Our coterm type \( T \uparrow T' \) is \( T'/\$.T \).

Our impure term type \( T \downarrow T' \) is \( T/\$.T' \).
Alternate between refocusing and reducing.

\[ \# \: "Goldilocks said: \ " \sim \\
(\# \: "This porridge is \ " \sim "too hot" \sim ".") \]

\[ = \#; ("Goldilocks said: \ " \sim) \$
(\#; ("This porridge is \ " \sim) \$ "too hot" \sim ".") \]

\[ \sim \#; ("Goldilocks said: \ " \sim) \$
(\#; ("This porridge is \ " \sim) \$ "too hot. ") \]

\[ = \#; ("Goldilocks said: \ " \sim) \$ (\# \: "This porridge is \ " \sim "too hot. ") \]

\[ \sim \#; ("Goldilocks said: \ " \sim) \$ (\# \: "This porridge is too hot. ") \]

\[ = \# \: "Goldilocks said: \ " \sim "This porridge is too hot. ") \]

\[ \sim \# \: "Goldilocks said: This porridge is too hot. ") \]

\[ = "Goldilocks said: This porridge is too hot. " \]
## Shift: dynamic semantics

\[
\begin{align*}
\# & \quad \$ \quad \text{“Goldilocks said: ” } \nabla \\
& \quad \$ \quad \text{“This porridge is ” } \nabla \\
& \quad \text{(出} k. (k \$ \text{“too hot”}) \nabla (k \$ \text{“too cold”}) \nabla (k \$ \text{“just right”})) \\
& \quad \nabla \text{“.”} \\
= \#; (\text{“Goldilocks said: ” } \nabla) \$ \\
& \quad (\text{“.”, } (\#; (\text{“This porridge is ” } \nabla))); \nabla) \$ \\
& \quad (\text{出} k. (k \$ \text{“too hot”}) \nabla (k \$ \text{“too cold”}) \nabla (k \$ \text{“just right”})) \\
\sim \quad \#; (\text{“Goldilocks said: ” } \nabla) \$ \# \$ \\
& \quad (((\text{“.”, } (\#; (\text{“This porridge is ” } \nabla))); \nabla) \$ \text{“too hot”}) \nabla \\
& \quad (((\text{“.”, } (\#; (\text{“This porridge is ” } \nabla))); \nabla) \$ \text{“too cold”}) \nabla \\
& \quad (((\text{“.”, } (\#; (\text{“This porridge is ” } \nabla))); \nabla) \$ \text{“just right”})) \\
= \#; (\text{“Goldilocks said: ” } \nabla) \$ \# \$ \\
& \quad ((\# \$ \text{“This porridge is ” } \nabla \text{“too hot” } \nabla \text{“.”})) \nabla \\
& \quad ((\# \$ \text{“This porridge is ” } \nabla \text{“too cold” } \nabla \text{“.”})) \nabla \\
& \quad ((\# \$ \text{“This porridge is ” } \nabla \text{“just right” } \nabla \text{“.”})) \\
\sim \quad \ldots
\end{align*}
\]
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” ∼ “said the porridge is too hot.”
Anaphora and context marks

(“Goldilocks” ∼) (#$“ said the porridge is too hot.”) 
∼ “Goldilocks said the porridge is too hot.”
Anaphora and context marks

(\texttt{interp \textasciitilde \textquote{Goldilocks}})(\texttt{\# $\text{String \textasciitilde said the porridge is too hot.}$})

\texttt{interp \texttt{str} = function}
\texttt{  | String \texttt{x} \texttt{\rightarrow str \textasciitilde x}}
Anaphora and context marks

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

interp str = function
  | String x -> str \~ x
Anaphora and context marks

(interp "Goldilocks")
(#$“ said the porridge is too hot ” ∨ “for ” ∨
(出$k. Req(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)) \$’’
)

∽

((interp "Goldilocks") (# $ Req(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 " \sim "for " \sim
 (k. Req(Female, k)) \sim ".")
)

\sim

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

\sim

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

interp str = function
| String x -> str \sim x
| Req(Female, k) -> interp str (k $ str)
Anaphora and context marks

(\text{interp} \text{ "Goldilocks"})

\((\#\$ \text{ " said the porridge is too hot " } \sim \text{ " for " } \sim \)\)

\((\text{out}\ k. \text{Req} (\text{Female} , k)) \sim \text{ "."})\)

\(\sim\)

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

\(\sim\)

(\text{interp} \text{ "Goldilocks"})

\((\# \$ \text{ " said the porridge is too hot " } \sim \text{ " for " } \sim \text{ "Goldilocks"} \sim \text{ "."})\)

\(\sim\) “Goldilocks said the porridge is too hot for Goldilocks.”

\text{interp str = function}\n
\begin{verbatim}
| String x -> str \sim x
| Req(Female,k) -> interp str (k \$ str)
\end{verbatim}
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)
(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 Female “Goldilocks”)
(♯ $ “ tasted ” ⊸ (interp Thing “the porridge”)
(♯ $ Req(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

\sim\rightarrow

(interp Female "Goldilocks")

(\# $ "tasted" \land((interp Thing "the porridge")

(\# $ "and said that" \land "the porridge"

"is too hot for" \land (出k. Req(Female, k)) \land ".")))

interp mytag str = function

  | String x -> str \land x
  | Req(tag,k) when tag = mytag ->
    interp mytag str (k $ str)
Several Pronouns, Several Marks

\[\sim\]

(\text{interp Female "Goldilocks"})

(\# $ "tasted" \sim ((\text{interp Thing "the porridge"})

(\# $ "and said that the porridge is too hot for" \sim

(\text{出k. Req(Female, k)} \sim ".")))

\text{interp mytag str} = \text{function}

\mid \text{String } x \rightarrow \text{str } \sim x

\mid \text{Req(tag,k) when tag = mytag} \rightarrow

\text{interp mytag str (k } \$ \text{ str)}
Several Pronouns, Several Marks

(Intent Female "Goldilocks")

(\# $ " tasted " \sim ((interp Thing "the porridge")

(\# $ " and said that the porridge is too hot for " \sim

(\k. Req(Female, k) \sim ".") )))

(Intent Female "Goldilocks")

(\# $ " tasted " \sim ((interp Thing "the porridge")

(\# $ Req(Female, k_2))))

interp mytag str = function

| String x -> str \sim x
| Req(tag, k) when tag = mytag ->
  interp mytag str (k $ str)
| Req(tag, k) ->
  let v = \k. Req(tag, k) in interp mytag str (k $ v)
Several Pronouns, Several Marks

\[\leadsto\]

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

\text{interp mytag str = function}

\(| \text{String x - } \to \text{ str } \leadsto x \)

\(| \text{Req(tag,k) when tag = mytag } - \to \)
\text{ interp mytag str (k $ str) \)

\(| \text{Req(tag,k) } - \to \)
\text{ let v = 出}k. \text{Req(tag,k) in interp mytag str (k $ v)} \)
Several Pronouns, Several Marks

\[
\rightsquigarrow \\
(\text{interp Female "Goldilocks"}) \\
\left(\# \$ \"tasted\" \right) \rightsquigarrow \\
(let \ v = \text{pk}. \text{Req}(\text{Female}, k) \text{ in} \\
\text{interp Thing \"the porridge\"}(k_2 \$ \nu))
\]

\[
\rightsquigarrow \\
(\text{interp Female "Goldilocks"}) \\
(\# \$ \text{Req}(\text{Female}, k_3))
\]

interp mytag str = function
| String x -> str ◼ x
| Req(tag,k) when tag = mytag ->
    interp mytag str (k $ str)
| Req(tag,k) ->
    let \ v = \text{pk}. \text{Req}(\text{tag}, k) \text{ in} \text{interp mytag str} (k \$ \nu)
Several Pronouns, Several Marks

\[ \sim \]

(\text{interp} \text{ Female "Goldilocks"})

(\# \$ "tasted" \sim

(let v = "Goldilocks" in

interp \text{ Thing} "the porridge" (k_2 \$ v)))

interp mytag str = function
  | String x -> str \sim x
  | Req(tag,k) when tag = mytag ->
    interp mytag str (k \$ str)
  | Req(tag,k) ->
    let v = \llbracket k. Req(tag,k) \rrbracket in interp mytag str (k \$ v)
Several Pronouns, Several Marks

\[\sim\]

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

\[\sim\]

\text{interp mytag str = function}
\mid \text{String x -> str} \sim x
\mid \text{Req(tag,k) when tag = mytag ->}
\quad \text{interp mytag str (k \$ str)}
\mid \text{Req(tag,k) ->}
\quad \text{let v = \text{out} k. Req(tag,k) in interp mytag str (k \$ v)}
Goldilocks tasted the porridge and said that the porridge is too hot for Goldilocks.

```latex
interp mytag str = function
    | String x -> str \texttt{\_\_} 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 = 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)
〜 this · (is(λe. 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. e · (f 「e」) · e))」
let だ f = fun e -> make_app e 「(is(λe. e · (f 「e」) · 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

\sim (\lambda x. this \cdot (is (\lambda e. e \cdot x)))

let (\cdot) x f = f x
let make_app x f = x \langle \cdot \rangle \langle \cdot \rangle f
let これは = 「this」
let 本 e = make_app e 「a-book」
let です f = fun e -> make_app e \langle (is (\lambda e. \langle \cdot \rangle (f \langle e \rangle \langle \cdot \rangle)) \langle \cdot \rangle) \rangle
let だ f = fun e -> make_app e \langle (is (\lambda e. \langle \cdot \rangle (f \langle e \rangle \langle \cdot \rangle)) \langle \cdot \rangle) \rangle
let 何 = 出k. \langle (\lambda x. \langle \cdot \rangle (k \$ (\lambda e. make_app e \langle x \rangle)) \langle \cdot \rangle) \rangle
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(λe. 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

\( \sim (\lambda x.(\text{this} \cdot (\lambda e. e \cdot x))) \cdot \text{so-he-said} \)

let (·) x f = f x
let make_app x f = x  \cdot  \text{·}  \cdot  f
let \text{これは} = 「this」
let \text{本 e} = make_app e 「a-book」
let \text{です f} = \text{fun e \rightarrow make_app e 「(is(\lambda e. \cdot \cdot \cdot (f \cdot e) \cdot \cdot \cdot))」}
let \text{だ f} = \text{fun e \rightarrow make_app e 「(is(\lambda e. \cdot \cdot \cdot (f \cdot e) \cdot \cdot \cdot))」}
let \text{何 = 出k. 「(\lambda x. \cdot \cdot \cdot (k \cdot (\lambda e. \text{make_app e} \cdot \cdot \cdot )) \cdot \cdot \cdot)」}
let \text{か f} = \# $ f
let と言いました f = make_app (「(\cdot \cdot \cdot f() \cdot \cdot \cdot)」) 「so-he-said」
Generating denotations of questions

(((これは・(何・だ))・言いました)・か

〜 (λx.(this • (is(λe.e • x))) • so-he-said)

(((これは・(何・です))・か)・言いました)

let (·) x f = f x
let make_app x f = x • [this]
let これは = 「this」
let 本 e = make_app e 「a-book」
let です f = fun e -> make_app e 「(is(λe.e • (f • e))」
let だ f = fun e -> make_app e 「(is(λe.e • (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
〜 (λx.(this · (is(λe.e · x)))) · so-he-said)
(((これは・(何・です))・か)・と言いました)
〜 (λx.(this · (is(λe.e · x)))) · so-he-said

let (·) x f = f x
let make_app x f = x ^ 「this」
let これは = 「this」
let 本 e = make_app e 「a-book」
let です f = fun e -> make_app e 「(is(λe.e · (f 「e」) ·」」
let だ f = fun e -> make_app e 「(is(λe.e · (f 「e」) ·」」
let 何 = 出 k. 「(λx.(k $ (λe.make_app e 「x」)) ·」」
let か f = # $ f
let 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 × breakpt)

The type of 42:

- `int`
- `int → bool` → `bool`
- `int → α` → `α` : context independence
- `int → F` → `F`
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

\[ 1 < 2 \]

\[ \lambda k. (\lambda k. k1) (\lambda x. (\lambda k. k2) (\lambda y. k(x < y))) \]
Answer types in the CPS transformation

1 < 2

\[(\text{bool} \rightarrow T) \rightarrow T\]

\[(\text{int} \rightarrow T) \rightarrow T\]

\[(\text{int} \rightarrow T) \rightarrow T\]

\[\lambda k. (\lambda k. k1)\]

\[\lambda x. (\lambda k. k2)\]

\[\lambda y. k(x < y)\]

\[T\]

\[T\]

\[T\]
Answer types in the CPS transformation

1 < 2

\[
(\text{bool} \rightarrow T) \rightarrow T
\]

\[
\left( (\text{int} \rightarrow T) \rightarrow T \right) \rightarrow T
\]

\[
\lambda k. \left( (\text{int} \rightarrow T) \rightarrow T \right) \rightarrow T
\]

\[
\lambda x. (\lambda k. k1) \quad (\lambda y. k(x < y))
\]

\[
T
\]

\[
T
\]
Answer types in the CPS transformation

\[ 1 < 2 \]
\[ (\text{出} k. \text{“Ouch!”}) < 2 \]

\[(\text{bool} \rightarrow T) \rightarrow \text{string} \]
\[\frac{\text{(int} \rightarrow T) \rightarrow \text{string}}{\lambda k. (\lambda k. \text{“Ouch!”}) (\lambda x. (\lambda k. k2) (\lambda y. k(x < y) T)) T \text{string}} \]
Answer types in the CPS transformation

\[ 1 < 2 \]
\[ (\text{出} k. \text{“Ouch!”}) < 2 \]
\[ 1 < (\text{出} k. \text{‘c’}) \]

This diagram represents the answer types in the CPS transformation.
Answer types in the CPS transformation

1 < 2

(出$k$. “Ouch!”) < 2

1 < (出$k$. ‘c’)

(出$k$. “Ouch!”) < (出$k$. ‘c’)

\[(\text{bool} \rightarrow T) \rightarrow \text{string}\]

\[(\text{int} \rightarrow \text{char}) \rightarrow \text{string} \quad \text{(int} \rightarrow T) \rightarrow \text{char}\]

\[\lambda k. \quad (\lambda k. \text{“Ouch!”}) \quad (\lambda x. \quad (\lambda k. \text{‘c’}) \quad (\lambda y. \quad k(x < y) \quad ))\]

\[\text{char}\]

\[\text{string}\]

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