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

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

I am very grateful indeed to Professor Yoshimoto and to Rui Otake for organizing this meeting and giving me the opportunity to talk on my favorite subject.

?

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`

To my shame I do not know the audience, and so I biased the talk to be motivational rather than formal. Another bias of the talk is towards computation (evaluation), even when analyzing linguistic phenomena. I am a Computer Scientist, and this bias is hard to overcome. Since continuations can be rather tricky, albeit only superficially so, it might not be immediately clear that my points make sense. Please do interrupt me and ask to clarify. If confusion is allowed to persist, it will only grow.

Outline

- **Delimited continuations**

Examining the stack

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

CPS and types

Summary

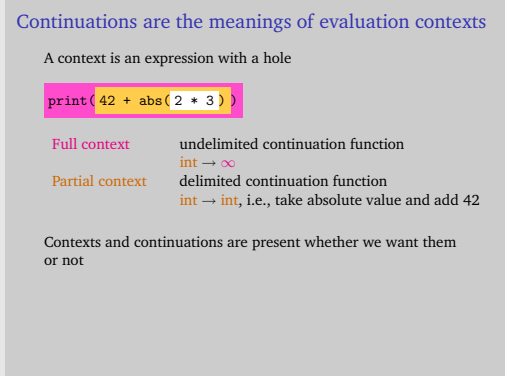
Continuations are the meanings of evaluation contexts

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



Before considering linguistic applications, let us talk about continuations in the broader context of Computer Science, where they may be easier to describe.This print expression is the *whole* program, which we want to run. To this end, we first *focus* (technical term) on the (sub)expression $2 * 3$ so to compute it first. If we *cut* this expression from the program, what is left is a program with the hole. The hole is the place where $2 * 3$ used to be and which we later fill with the result of evaluating $2 * 3$. The expression with the hole is called *context*. The undelimited continuation is the meaning of the context. It is a function from what me may put in the hole (integers in our case) to . . . well, the result of the whole program. This is what computed when the whole program is fully finished – and so this value is not of much interest to the program itself as the program will never get to use this value.

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 → ∞
Partial context	delimited continuation function int → int, i.e., take absolute value and add 42

Contexts and continuations are present whether we want them or not

When the result is computed, the program is already dead. For example, we usually don't care of the value computed by our e-mail program. We are much more interested in what the e-mail program does before it finishes or dies (i.e., has it sent the message or not).

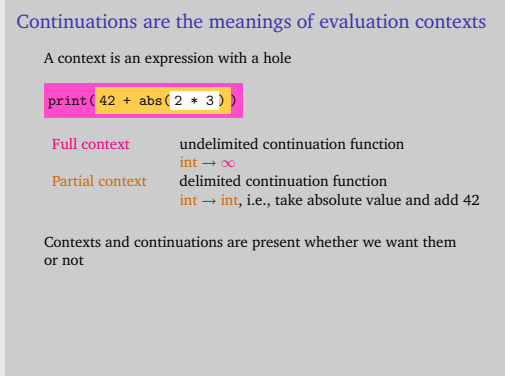
Continuations are the meanings of evaluation contexts

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

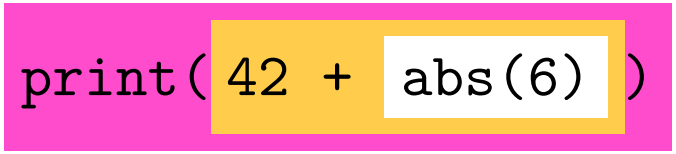
Contexts and continuations are present whether we want them or not



Beside the full context, we may also want to consider its prefix. That is, we may (mentally, for now) distinguish a subterm of a program, $42 + abs(2 * 3)$. We may imagine a boundary within print(). Taking out $2 * 3$ leaves a hole in our subterm just as it did in the whole program. This subterm with a hole is called a *partial* (evaluation) context, whose meaning is a *partial* continuation. (The subterm with a hole can be plugged into a bigger hole). The partial continuation is also a function, also from integers in our case (the type of the values that can be placed in the hole, e.g., the result of evaluating $2 * 3$). Now, however, we do care of the produced result (also called the *answer*), since we *can* do something meaningful with it: plug into a hole. So, the delimited continuation in our case is a function from int to int, namely, the function that takes an integer and adds to its absolute value 42.

Continuations are the meanings of evaluation contexts

A context is an expression with a hole



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

Let us observe what happens with the partial context as we are evaluating the term. We see the context shrinks as subterms are reduced and are replaced with values.

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 6>0 then 6 else neg(6) )
```

Contexts and continuations are present whether we want them or not

We also see the partial context expand when functions are invoked and their bodies are inlined.

Continuations are the meanings of evaluation contexts

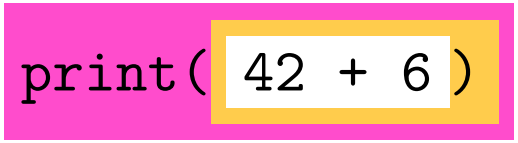
A context is an expression with a hole

```
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

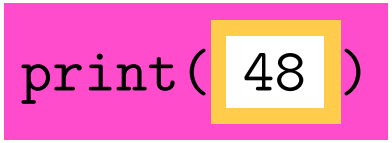
A context is an expression with a hole



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



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

Finally, our distinguished subterm is reduced to a single value and is no longer useful to distinguish it. Nothing can ever happen to 48.

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

So, the boundary and the yellow thing it envelops disappear. I apologize for the triviality of all of this. Things will become more complex very soon. We will also make the notions of disappearing boundaries and plugging of the hole precise.

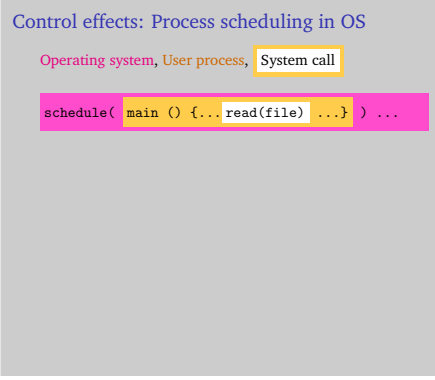
Whether we are concerned with the continuations or not, they are *always* present.

(Delimited) continuations are the meanings of (delimited) evaluation contexts.

Control effects: Process scheduling in OS

Operating system, User process, System call

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



Let us consider a different example: the OS has invoked a user process, and the process is about to make a system call, that is, request the OS, the supervisor, to read from a given file. The slide shows the state of the whole program at this point.

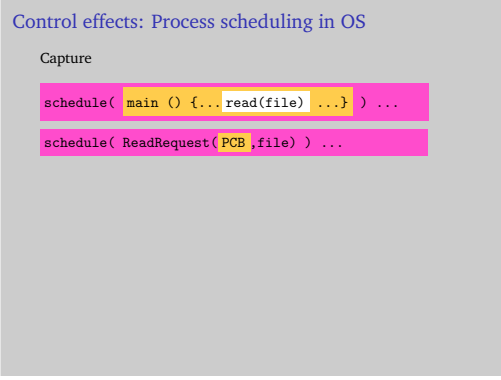
Now it makes great sense to distinguish the subterm that represents the user program (in yellow) from the rest. This is the kernel-user boundary. This example also makes it clear why we usually don't care of the result of the whole program: when the OS returns a result it is because it crashed, at which point we quickly reboot.

Control effects: Process scheduling in OS

Capture

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

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



We do one step of evaluation, and see the different picture from the one in the earlier example. All spread-out yellow stuff has disappeared in one step, replaced with this term `ReadRequest(PCB,file)`. Such a behavior is characteristic of *control effects*. But the yellow stuff is not gone, it is somehow 'saved' in the value we call here PCB, or, the process control block in the OS parlance. How the context is saved is not important for us now. We only need to know that the saved context can be restored.

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

```
schedule( main () {... read(file) ...} ) ...
schedule( ReadRequest(PCB,file) ) ...
...
schedule( resume(PCB,"read string") ) ...
```

When the OS gets around to reading from a file, it does the following operation. The next reduction is:

Control effects: Process scheduling in OS

Capture, Invoke

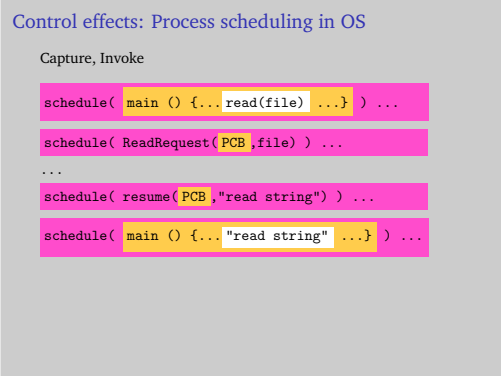
```
schedule( main () {... read(file) ...} ) ...

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

...

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

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



We get the picture very similar to the original one, only with `"read string"` in place of `read(file)`.

We have seen thus the two control operations on the contexts: *capturing* a context (saving it in a value like PCB) and *restoring* it. The latter operation takes the captured context (`PCB`), a value (”read string”), plugs the value into the saved context and puts the result in the context of the restoring operation. It is easy to see that this context invocation looks exactly like the function application (cf the invocation of `abs(6)` in the first example).

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 $\Rightarrow$ user-level scheduling, thread library

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

The operations of capturing and resuming are obviously special, here in the sense that only OS can do them. One may imagine context capturing and restoring available to user programs, too. We can then implement user-level threads and write scheduling libraries.

Captured continuation can be invoked once, none, or several times.

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 ) )
```

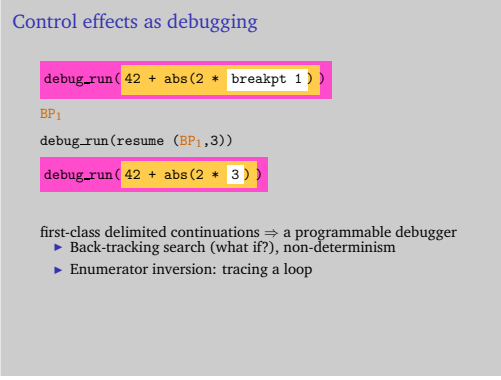
BP_1

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

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

first-class delimited continuations $\Rightarrow$ a programmable debugger

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



We note that we don't have to resume from the breakpoint at all. But we did execute `resume (BP1,3)`, which restored the context, replaced the breakpoint expression with 3, and continued running the program. Suppose we don't like the computed result, 48 in our case. We still possess the captured continuation saved as BP_1. We can resume it again, with a different value. So, we can do backtracking and implement non-determinism.

If the context capturing and restoring were available to the user program rather than to the debugger only, our program could support non-determinism. We could also trace a loop (aka the enumerator inversion, ref. the CONTEXT poster), to pace it.

Reset

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

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

Reset

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

```
  # $ "Goldilocks said: " ⌢
    (# $ "This porridge is " ⌢ "too hot" ⌢ ". ")
⇝ # $ "Goldilocks said: " ⌢ (# $ "This porridge is " ⌢ "too hot. ")
⇝ # $ "Goldilocks said: " ⌢ (# $ "This porridge is too hot. ")
⇝ # $ "Goldilocks said: " ⌢ "This porridge is too hot. "
⇝ # $ "Goldilocks said: This porridge is too hot. "
⇝ "Goldilocks said: This porridge is too hot. "
```

A brief explanation of the tale "Goldilocks and the Three Bears" and how come Goldilocks tasted the porridge, and what happened next. Goldilocks managed to escape.

If operations to capture and restore contexts were available to user programs, what form would they take? We describe the most common control operators, shift and reset. We start with reset. We introduce a special infix operator, called plug, $, which takes a context on the left and a term on the right, and *represents* the filling of the hole in the context with the right argument. In order to use plug, we need some context for its left operator: we define #, as the trivial context, made of the hole only. Filling the hole with the value v gives us back v no matter what v is.

The evaluation, better done by hand, illustrates two things: evaluating terms in the presence of $ and eliminating the $.

Shift

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

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

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

Shift

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

```
# $ "Goldilocks said: " ⌢
  (# $ "This porridge is " ⌢
    (出k.(k $ "too hot") ⌢ (k $ "too cold") ⌢ (k $ "just right"))
                                                    ⌢ ". ")
⇝ # $ "Goldilocks said: " ⌢
    (# $ ((# $ "This porridge is " ⌢ [] ⌢ ". ") $ "too hot") ⌢
         ((# $ "This porridge is " ⌢ [] ⌢ ". ") $ "too cold") ⌢
         ((# $ "This porridge is " ⌢ [] ⌢ ". ") $ "just right"))
```

shift captures the context up to and including the closest $ and # behind it.

Shift

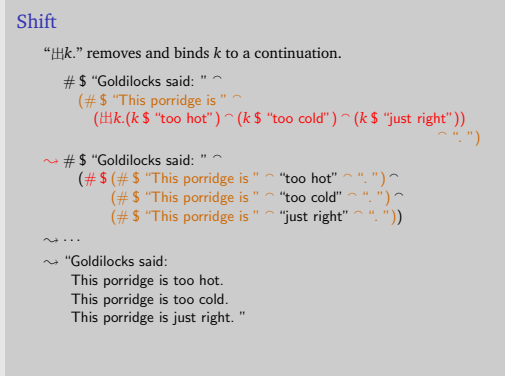
"出k." removes and binds k to a continuation.

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

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

$\rightsquigarrow \cdots$

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



shift replaces the captured context with #$ followed by its body, with all occurrences of *k* being replaced by the captured context.

Why we need to insert a plug before the body of the shift in the replacement of the captured context: to serve as a delimiter, the closest plug, in case the body of shift includes other shifts. The plug in the captured continuation will act as the closest plug when the continuation is restored. So, we are statically assured that "Goldilocks said" will never be in the captured continuation no matter what the body of the shift is or does.

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

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

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

Terms	$E, F ::= V \mid FE \mid C\,\$\,E \mid \text{出}k.E$
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	

$$C_1\,\$\cdots\$\,C_n\,\$\,(\lambda x.E)V \quad \rightsquigarrow C_1\,\$\cdots\$\,C_n\,\$\,E\{x \mapsto V\}$$
$$C_1\,\$\cdots\$\,C_n\,\$\,C\,\$\,(\text{出}k.E) \rightsquigarrow C_1\,\$\cdots\$\,C_n\,\$\,\#\,\$\,E\{k \mapsto C\}$$

Now we formalize the hand-waving about the hole and its filling. It is nice to be fully explicit once in a while.
If we disregard the red stuff, the calculus is the familiar simply-typed lambda-calculus.

Note that some terms are syntactically classified as values. Note the symmetry in the reductions rules for lambda and shutu.

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 $T\backslash_{\$}T'$.

Structural rules express evaluation order

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

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

Our coterm type $T \uparrow T'$ is $T'/_{\$}T$.
Our impure term type $T \downarrow T'$ is $T\backslash_{\$}T'$.

The structural rules are *equalities* and can be applied in any order.

Reset: dynamic semantics

Alternate between refocusing and reducing.

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

Shift: dynamic semantics

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

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

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

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

$$
\rightsquigarrow \cdots
$$

Outline

Dynamic binding: summary

Many applications

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

If the function dynamically binds the *current* (working directory, locale, etc) binding, the binding is available not only inside the function but also in every invoked function. The absence of closure with dynamic binding is characteristic.

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 for her.

As we are taught in school: to resolve a pronoun, look *through the context* for the appropriate noun.

Anaphora and context marks

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

Anaphora and context marks

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

$\rightsquigarrow$ “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")(# $ String " said the porridge is too hot.")

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

We will drop the String to ease the notation

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 ” ⌢ “for ”⌢
(出k. Req(Female, k)) ⌢ “.”)

⇝

(interp “Goldilocks”)(# $ Req(Female, k))

⇝

(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 ” ⌢ “for ”⌢
(出k. Req(Female, k)) ⌢ “.”)

⇝

(interp “Goldilocks”)(# $ Req(Female, k))

⇝

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

⇝ “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

(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)) ⌢ “.”)))

$\rightsquigarrow$

(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

$\rightsquigarrow$

(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

$$\rightsquigarrow$$

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

$$\rightsquigarrow$$

$$(\text{interp}\ \text{Female}\ \text{“Goldilocks”})\ (\#\ \$\ \text{“ tasted ”} \frown ((\text{interp}\ \text{Thing}\ \text{“the porridge”})\ (\#\ \$\ \text{Req}(\text{Female}, k_2))))$$

```
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

$\rightsquigarrow$

(interp Female "Goldilocks")
  (# $ " tasted "$\frown$
    (*let* $v = \text{出}k.\,\text{Req}(\text{Female}, k)$ *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

$$\rightsquigarrow$$

$$(\text{interp Female “Goldilocks”})$$
$$\quad (\#\ \$\ \text{“ tasted ”} \frown$$
$$\quad\quad (\mathit{let}\ v = \text{出}k.\ \text{Req}(\text{Female}, k)\ \mathit{in}$$
$$\quad\quad \text{interp Thing “the porridge”}\ (k_2\ \$\ v)))$$

$$\rightsquigarrow$$

$$(\text{interp Female “Goldilocks”})$$
$$\quad (\#\ \$\ \text{Req}(\text{Female}, k_3))$$

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

Several Pronouns, Several Marks

$\rightsquigarrow$

(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

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 = μ̂k. 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

The `Req` clause represents the case when I received your request and I forward it to the supervisor, if *I* so chose. The `ReqDefer` clause represents the case when you received my answer, presumably are dissatisfied with it, and *you* choose to see my supervisor.

For `ReqDefer` approach, see our ICFP06 paper. The forwarding `Req` trick is the intuition behind one of the proofs that shift/reset can emulate control/prompt.

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

Generating denotations of questions

これは · (本 · です)
$\rightsquigarrow$ *this* · (*is*(λ*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 · (本 · です)
⇝ *this* · (*is*(λ*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)
```

We see a familiar phrase, only with some weird dots and parentheses. But this phrase is a program expression, and it can *evaluated* as a program, given the appropriate definitions for です, etc. The evaluation gives a *denotation* – currently a string, but we have more sophisticated denotational domains. In the current example, it is clear the denotation is well typed, see the types of 'this', etc. later. It is not obvious that this is always the case. We can nevertheless statically assure that only well-typed denotations are produced, see our final tagless paper.

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 ^ 「·」 ^ 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
```

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

Generating denotations of questions

```
(これは · (何 · です)) · か
⇝ (λx. this · (is(λe. e · x)))

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


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

Before we used shift to insert something inside. Now, we insert something outside, which corresponds to the natural style of writing. If we write an essay and realized that we need to use a term we should have defined earlier, we bookmark the current position in the Emacs buffer, go to the earlier part of the essay, insert the definition, come back to the remembered location and continue writing. The same applies when we write a Haskell program and realized we need to use a function defined in a module that we haven't imported. We have to jump the beginning of our code, insert the import statement, and come back to the current expression.

Generating denotations of questions

(これは · (本 · だ)) · と言いました
$\rightsquigarrow$ (*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
$\rightsquigarrow$ ($\lambda x.$(*this* · (*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」
```

Generating denotations of questions

((これは · (何 · だ)) · と言いました) · か
$\rightsquigarrow$ ($\lambda x.(this \cdot (is(\lambda e.\, e \cdot x))) \cdot$ 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
↝ (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 ^ 「·」 ^ 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
⇝ (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 ^「.」^ 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」
```

Point out how the placement of the control delimiter, か, affects the scope of the binding introduced by lambda. Thus we obtain either a question that includes a quotation, or a quotation that includes a question.

Outline

The following was skipped during the presentation. It was 1.5-hr long already. I have prepared a non-traditional presentation of CPS, using operations on contexts rather than on terms. I should write it up for my web site later.

Introduction to CPS

$42 < (2 \times breakpt)$

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

$$1 < 2$$

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

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{T}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{T}$$

Answer types in the CPS transformation

$$1 < 2$$

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

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{T}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{T}$$

Answer types in the CPS transformation

$$1 < 2$$

$$(出k.\ \text{“Ouch!”}) < 2$$

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

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{string}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{T}$$

Answer types in the CPS transformation

$$1 < 2$$

$$(\text{出}k.\ \text{“Ouch!”}) < 2$$

$$1 < (\text{出}k.\ \text{‘c’})$$

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

Answer types in the CPS transformation

$$1 < 2$$

$$(出k.\ \text{“Ouch!”}) < 2$$

$$1 < (出k.\ \text{‘c’})$$

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

$$\overbrace{\overbrace{(\text{int} \to \text{char}) \to \text{string}}^{(\text{bool} \to T) \to \text{string}} \quad \overbrace{(\text{int} \to T) \to \text{char}}^{\text{char}}}^{\text{string}}$$

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

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

Outline

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`