Recursive Programs in the Abstract

Masahito Hasegawa
RIMS, Kyoto University
PRESTO, JST

PPL2004, 13 March 2004

Semantics of Recursive Programs

A *recursive program* like

$$\mathsf{fact} \;\equiv\; \lambda x^{\mathsf{int}}.\,\mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ x \times \mathsf{fact}(x-1) \;:\; \mathsf{int} \to \mathsf{int}$$

can be understood as a *fixed point* of

$$F \;\equiv\; \lambda f^{\mathsf{int}\to\mathsf{int}}.\,\lambda x^{\mathsf{int}}.\,\mathsf{if}\ x = 0\ \mathsf{then}\ 1\ \mathsf{else}\ x \times f(x-1)$$
$$:\; (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$$

i.e. $\mathsf{fact} = F(\mathsf{fact})$.

Fundamental Idea: Recursive programs are modelled by *fixed points* on certain mathematical structures $\longrightarrow$ *Denotational Semantics*

$$[\![\mathsf{fact}]\!] = [\![F\,(\mathsf{fact})]\!] = [\![F]\!]\,([\![\mathsf{fact}]\!]) = \bigcup_{n=0}^{\infty} [\![F]\!]^n(\bot)$$

$$
\begin{array}{lcl}
 & & \mathsf{fact}(2) \\
(\mathsf{fact} = F(\mathsf{fact})) & = & F(\mathsf{fact})(2) \\
(\text{def. of } F) & = & (\lambda f.\lambda x.\text{if } x = 0 \text{ then } 1 \text{ else } x \times f(x-1))(\mathsf{fact})(2) \\
 & = & \text{if } 2 = 0 \text{ then } 1 \text{ else } 2 \times \mathsf{fact}(2-1) \\
(2 \neq 0) & = & 2 \times \mathsf{fact}(1) \\
(\mathsf{fact} = F(\mathsf{fact})) & = & 2 \times F(\mathsf{fact})(1) \\
(\text{def. of } F) & = & 2 \times (\text{if } 1 = 0 \text{ then } 1 \text{ else } 1 \times \mathsf{fact}(1-1)) \\
(1 \neq 0) & = & 2 \times (1 \times \mathsf{fact}(0)) \\
(\mathsf{fact} = F(\mathsf{fact})) & = & 2 \times (1 \times F(\mathsf{fact})(0)) \\
(\text{def. of } F) & = & 2 \times (1 \times (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 \times \mathsf{fact}(0-1))) \\
(0 = 0) & = & 2 \times (1 \times 1) \\
 & = & 2
\end{array}
$$

Criticisms on the Traditional Approach

Nice mathematics, but overly simplified the computational aspects:
lots of sophisticated theories for toy languages which

- cannot explain how recursion is created
- cannot explain how recursion interacts with other features of programming languages

---

Wanted: right level of abstraction for studying these issues
(not too abstract, not too concrete –
cf. types, abstract interpretation, attribute grammar)

Overview of This Talk

Case studies on the semantics of recursive computation:

I Recursion from Cyclic Sharing

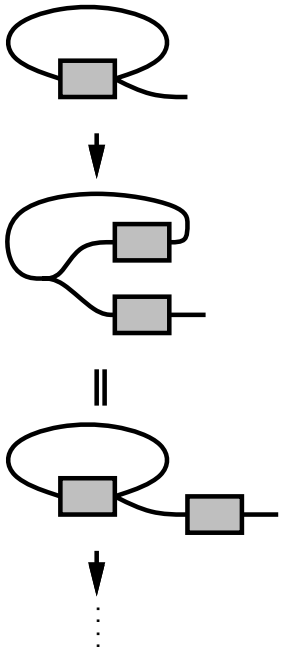
II Recursion and Control

III Recursion and Duality

. . . to convince you that the semantic (and abstract) approaches to programming languages are interesting and fruitful

Part I: Recursion from Cyclic Sharing

Recursion from Cyclic Structure (e.g. Turner, 70's)

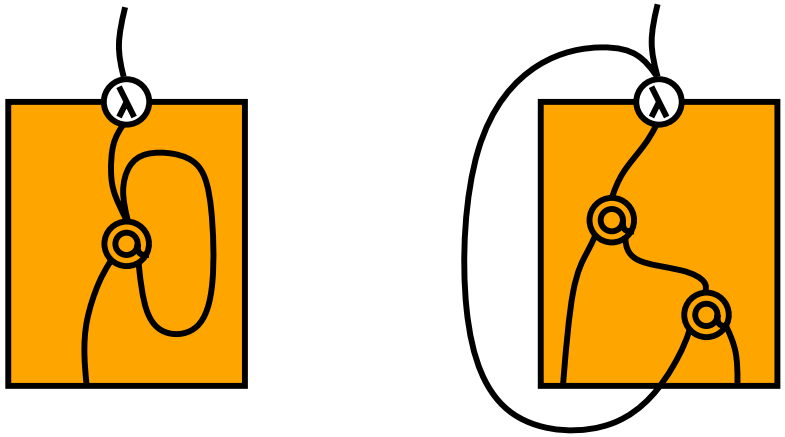


recursive call = copying of circularly shared resource

Wanted: Semantics which can explain the precise relationship between recursion and cyclic structures

Example: Cyclic Lambda Graphs

(scoped) λ-graphs for fixpoint computation



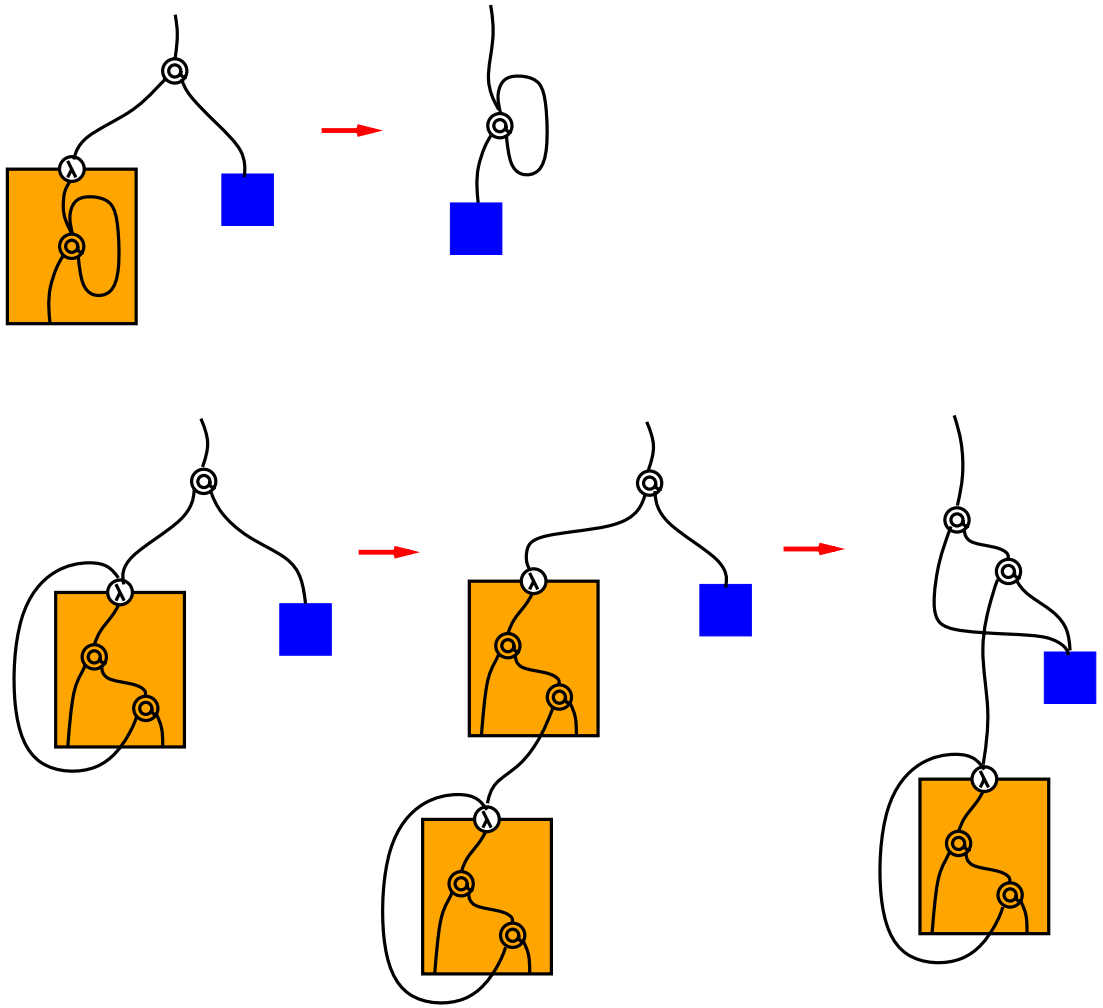
$\lambda f.$letrec x be fx in x $\stackrel{?}{=}$ letrec F be $\lambda f.f(Ff)$ in F

cf. Church's combinator

$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

cf. Turing's combinator

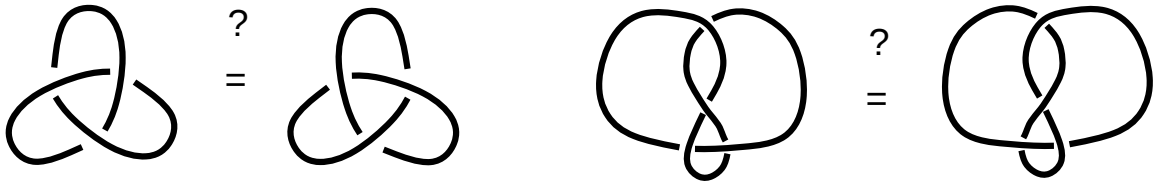
$\Theta = Y(\lambda F.\lambda f.f(Ff))$



(cf. Launchbury, POPL'93)

Analogy to Knot Theory

Mathematicians ask *if two knot diagrams represent the same knot*



and study their algebraic interpretations (knot invariants)

Computer scientists ask *if two programs behave in the same way*
and study their *semantics*

Example: Jones-Conway Polynomial (with two variables)

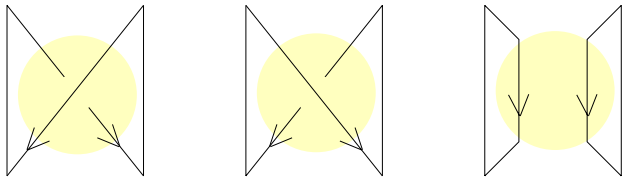
There exists a unique map P from the set of all oriented links in $\mathbb{R}^3$ to the ring $\mathbb{Z}[x, x^{-1}, y, y^{-1}]$ such that

(i) (*soundness*) if $L \sim L'$ then $P(L) = P(L')$

(ii) P maps the trivial knot to 1

(iii) if (L_+, L_-, L_0) is a *Conway triple* then

$$xP(L_+) - x^{-1}P(L_-) = yP(L_0).$$



Example of Conway Triple

In particular: L, L' mirror images $\Rightarrow P(L')(x, y) = P(L)(x^{-1}, y^{-1})$

Though *not* complete, P helps to distinguish the trefoil knots in the last slide

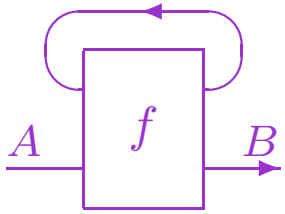
Traced Monoidal Categories as Models of Cyclic Structure

(Joyal, Street and Verity, 1996)

Categorical structure for cyclic structures like knots and cyclic graphs

Technically: a *traced monoidal category* is a (balanced) monoidal category equipped with a *trace* operator which creates a loop

$$\frac{f : A \otimes X \to B \otimes X}{Tr^X_{A,B}(f) : A \to B}$$

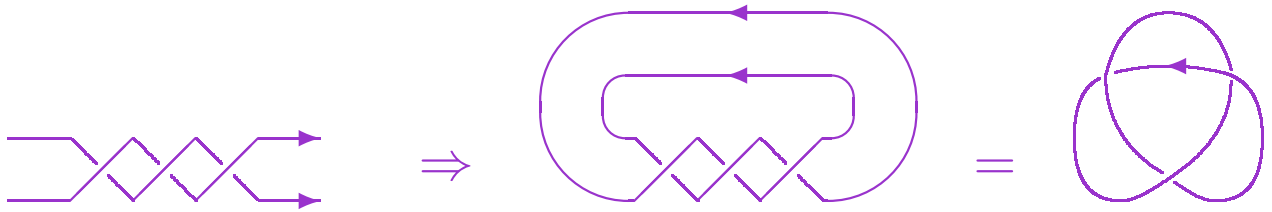


subject to a few coherence axioms.

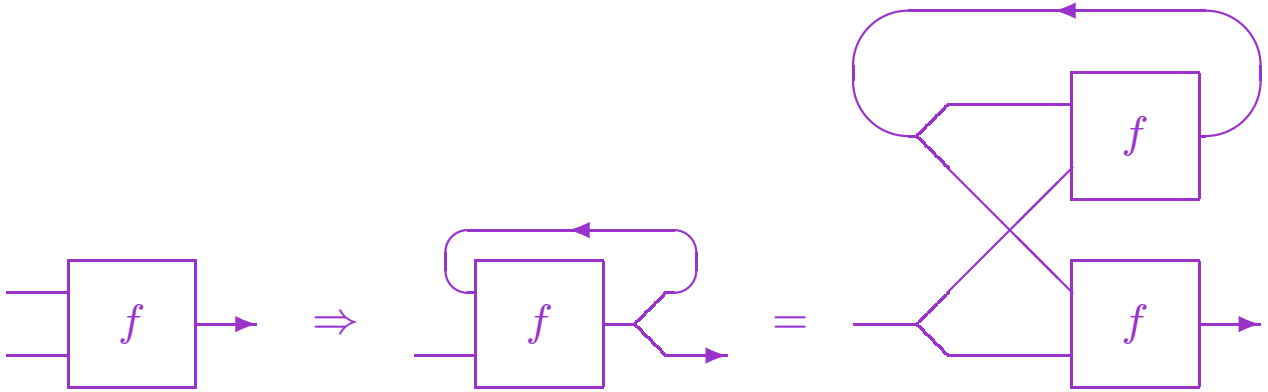
$\longrightarrow$ use traced monoidal categories for modelling cyclic (λ-)graph rewriting systems (Hasegawa, TLCA'97 / PhD Thesis, 1997)

Knots and Recursion via Traces

Trefoil Knot via Trace (Braid Closure):



Recursion via Trace (Cyclic Sharing):



Examples of Traced Monoidal Categories

Linear Algebra

The category of fin. dim. vector spaces and linear maps. For a linear map $f : U \otimes_K W \to V \otimes_K W$, its trace $Tr^W_{U,V}(f) : U \to V$ is given by

$$(Tr^W_{U,V}(f))_{i,j} = \Sigma_k f_{i\otimes k, j\otimes k}$$

Binary Relations

The category of sets and binary relations. For a relation $R : A \times X \to B \times X$ the trace $Tr^X_{A,B}(R) : A \to B$ is given by

$$Tr^X_{A,B}(R) = \{(a, b) \mid \exists x \in X\, (a, x)R(b, x)\}$$

Quantum Invariants of Knots

The category of representations of a quasi-triangular Hopf algebra.

Recursion from Cyclic Sharing in the Abstract

Under certain conditions, traces give rise to fixed-point operators:

Thm. (*Trace-Fixpoint Correspondence*, Hyland / Hasegawa) If the tensor product $\otimes$ is cartesian, there is a bijective correspondence between traces and dinatural diagonal (i.e. well-behaved) fixed-point operators.

Example: traditional domain theoretic models

Non-example: cyclic λ-graphs – covered by the generalisation below:

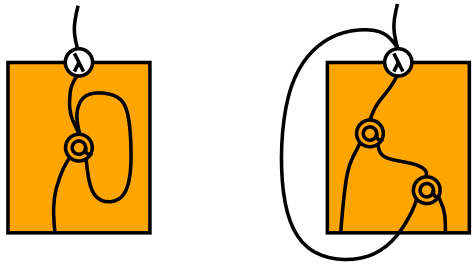
Thm. If there is a trace (for modelling cycles) and suitable adjunction from a cartesian category (for modelling (weak) λ-abstraction and application), there exists a dinatural fixed-point operator (and many other fixed-point like operators).

$\longrightarrow$ new semantic models for cyclic λ-graphs

Semantics of Cyclic Lambda Graphs

Thm. The structure described in the last theorem provides a sound and complete class of models for the cyclic lambda calculus.

Example: In the category of sets and binary relations
(the non-deterministic interpretation):



fix1 $= \lambda f.$letrec x be fx in x fix2 $=$ letrec F be $\lambda f.f(Ff)$ in F

$[\![\text{fix1}\ M]\!] = \bigcup_{f \in [\![M]\!]} \{x \mid (x,x) \in f\}$ $[\![\text{fix2}\ M]\!] = \bigcup_{f \in [\![M]\!]} \bigcup_{f \circ A = A} A$

fix1 works for function closures only (but more efficient), whereas
fix2 for any value (and less efficient) – hence $[\![\text{fix1}]\!] \subset [\![\text{fix2}]\!]$

Part II: Recursion and Control

Hasegawa, Sabry and Filinski

Recursion in Call-by-Value Languages

The naive fixpoint equation $\mathsf{fix}\, F \;=\; F\,(\mathsf{fix}\, F)$ cannot be justified!

To evaluate $\mathsf{fix}\, F$, we need to evaluate $\mathsf{fix}\, F$ in $F\,(\mathsf{fix}\, F)$

. . . does not stop

Right (well-known) solution:

$$\mathsf{fix}^{\mathsf{v}}\, F \;=\; \lambda x.F\,(\mathsf{fix}^{\mathsf{v}}\, F)\, x \quad (\text{or } \mathsf{fix}^{\mathsf{v}}\, F\, M \;=\; F\,(\mathsf{fix}^{\mathsf{v}}\, F)\, M)$$

. . . makes sense, but is it the *canonical* answer?

Any other better principles?

Example: Dinaturality

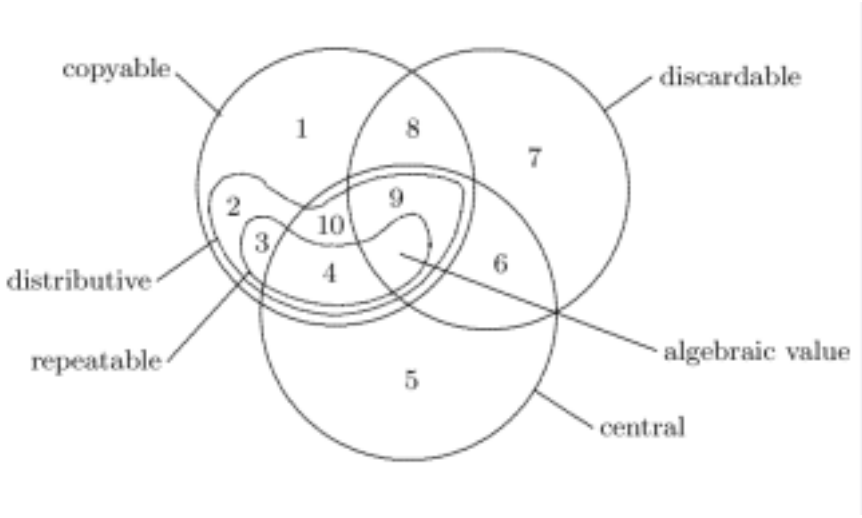
The *dinaturality* equation for CBN recursion

$$\mathsf{fix}\,(g \circ f) = g\,(\mathsf{fix}\,(f \circ g))$$

cannot be justified for CBV recursion:

```
- fun F (f:int->int) = (print "Hello\n"; fn x:int => x);
val F = fn : (int -> int) -> int -> int
- fun G (f:int->int) = (print "Bye\n"; fn x:int => x);
val G = fn : (int -> int) -> int -> int
- fix (G o F) 1;
Hello
Bye
val it = 1 : int
- G (fix (F o G)) 1;
Bye
val it = 1 : int
```

The Wild Nature of Call-by-Value (Führmann, FoSSaCS2002)



where M is called

a value if	let x be M in N	$=$	$N[M/x]$
copyable if	let x be M in (x, x)	$=$	(M, M)
discardable if	let x be M in N	$=$	$N \quad (x \notin FV(N))$
central if	let x be M in let y be N in L	$=$	let y be N in let x be M in L
			$(x \notin FV(N),\ y \notin FV(M))$

Semantics of Call-by-Value and Recursion

Semantics of call-by-value languages using monads (Moggi, LICS'89):
A monad T specifies the semantics of computational effects under consideration (e.g. non-determinism, exceptions, states, continuations)
TX denotes the type of *computation* on the type X

Semantics of call-by-value recursion via T-fixpoint operators
(Simpson and Plotkin, LICS2000):
Fixed-point operator with the type $(TX \to TX) \to TX$ subject to a uniformity axiom (analogous to that in domain theory)

Axioms for Recursion in Call-by-Value

(Hasegawa and Kakutani, FoSSaCS2001)

A type-indexed family of closed values $\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau} : ((\sigma \to \tau) \to \sigma \to \tau) \to \sigma \to \tau$ is called a *uniform call-by-value fixpoint operator* if the following conditions are satisfied:

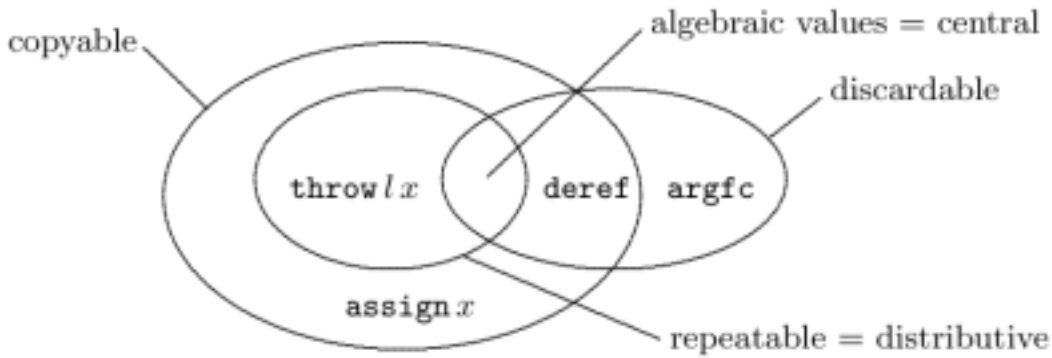
1. (CBV fixpoint) For any value $F : (\sigma \to \tau) \to \sigma \to \tau$
   $\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\, F \;=\; \lambda x^{\sigma}.F\,(\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\, F)\, x$
2. (stability) For any value $F : (\sigma \to \tau) \to \sigma \to \tau$
   $\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\, F \;=\; \mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\, (\lambda f^{\sigma\to\tau}.\lambda x^{\sigma}.F\, f\, x)$
3. (uniformity) For values $F : (\sigma \to \tau) \to \sigma \to \tau$, $G : (\sigma' \to \tau') \to \sigma' \to \tau'$ and $H : (\sigma \to \tau) \to \sigma' \to \tau'$,
   if $H(\lambda x.M\, x) = \lambda y.H\, M\, y$ holds for any $M : \sigma \to \tau$ and $H \circ F = G \circ H$ holds, then $H\,(\mathsf{fix}^{\mathsf{v}}_{\sigma\to\tau}\, F) \;=\; \mathsf{fix}^{\mathsf{v}}_{\sigma'\to\tau'}\, G$

Thm. Sound and complete for the semantics by Simpson and Plotkin.

Example: First-Class Continuations

(Thielecke, PhD Thesis, 1997 / Selinger, MSCS 11(2), 2001)

In the presence of *first-class continuations* (e.g. `callcc` in SML/NJ), the situation becomes rather simple:



Its semantics is given by a continuation monad $TX = (X \to R) \to R$.
In this setting, we can show that a uniform CBV fixed-point operator can be derived from a uniform CBV iterator, and vice versa – via Filinski's *"Recursion from Iteration"* construction.

Recursion from Iteration (Filinski '92/Hasegawa and Kakutani '01)

Let $\bot$ be an empty type (i.e. no closed value) and write $\neg\sigma$ for $\sigma \to \bot$.

A *CBV iterator*: $\mathsf{loop}_\sigma \;=\; \lambda f.(\mathsf{loop}_\sigma\, f) \circ f : (\sigma \to \sigma) \to \neg\sigma$

In loop f, f is evaluated repeatedly until something happens (as "while")

Thm. In the presence of first-class continuations, there is a bijective correspondence between uniform CBV fixpoint operators and uniform CBV iterators.

Recursion from Iteration in the Abstract

The essential reason why this theorem holds is that, in this setting, values, central terms, and discardable copyable central terms all *agree* (as shown in the last slide) – and this implies that uniform CBV iterators are modelled by T-fixed point operators in the semantic models, hence equivalent to uniform CBV fixpoint operators.

```
(* an empty type "bot" with an initial map A : bot -> 'a *)
datatype bot = VOID of bot;
fun A (VOID v) = A v;
(* the C operator, C : (('a -> bot) -> bot) -> 'a *)
fun C f =
 SMLofNJ.Cont.callcc (fn k => A (f (fn x => (SMLofNJ.Cont.throw k x) : bot)));

(* basic combinators *)
fun step F x = C (fn k => F k x);
fun pets f k x = k (f x) : bot;
fun switch l x = C (fn q => l (x,q));
fun switch_inv f (x, k) = k (f x) : bot;
(* step : (('a -> bot) -> 'b -> bot) -> 'b -> 'a
   pets : ('a -> 'b) -> ('b -> bot) -> 'a -> bot
   switch : ('a * ('b -> bot) -> bot) -> 'a -> 'b
   switch_inv : ('a -> 'b) -> 'a * ('b -> bot) -> bot *)

(* an iterator, loop : ('a -> 'a) -> 'a -> bot *)
fun loop f x = loop f (f x) : bot;

(* recursion from iteration *)
fun fix F = switch (loop (step (switch_inv o F o switch)));
(* fix : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b *)
```

Part III: Recursion and Duality

Kakutani and Hyland

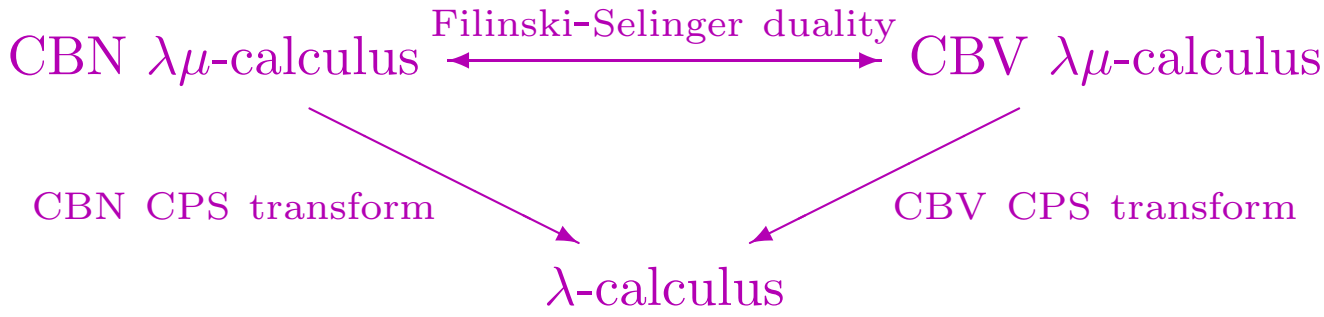
Selinger

The Filinski-Selinger Duality between CBN and CBV

(Filinski, CTCS'89 / Selinger, MSCS 11(2), 2001 / Wadler, ICFP2003)

Under the presence of first-class continuations, there exists a *duality* between call-by-name and call-by-value languages.

Syntactically:



Semantically, this duality amounts to the categorical duality between *control categories* (for CBN) and *co-control categories* (for CBV).

$\longrightarrow$ combine Filinski-Selinger duality with the categorical duality between recursion (fixed-point operator) and iteration

Duality between CBN Recursion and CBV Iteration

(Kakutani, CSL'02)

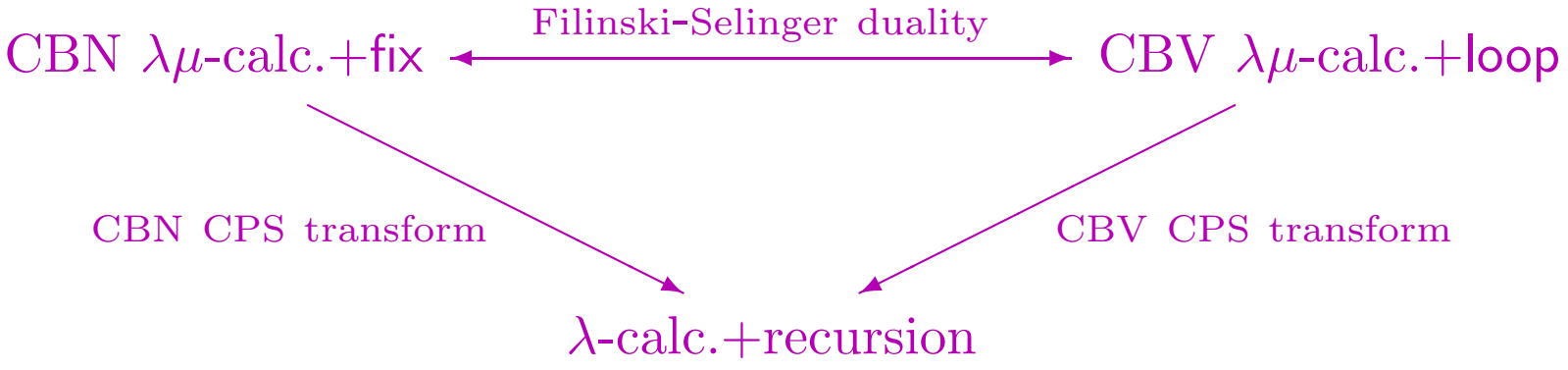
The duality between the CBN recursion

$$\mathsf{fix}\, f \;=\; f \circ (\mathsf{fix}\, f) \;=\; f \circ f \circ (\mathsf{fix}\, f) \;=\; \ldots$$

and the CBV iteration

$$\mathsf{loop}\, f \;=\; (\mathsf{loop}\, f) \circ f \;=\; (\mathsf{loop}\, f) \circ f \circ f \;=\; \ldots$$

is smoothly accommodated in the Filinski-Selinger duality:



Relating CBN Recursion and CBV Recursion

Under the presence of first-class continuations:

- In Part II, we have observed that there is a bijective correspondence between the CBV recursion and CBV iteration, subject to certain uniformity conditions (Filinski's "Recursion from Iteration").
- In the last slide, we have seen that there is a bijective correspondence between the CBN recursion and CBV iteration ("Filinski-Selinger Duality" extended by Kakutani).

$\longrightarrow$ By combining these two results, we obtain a correspondence between recursion in CBN and recursion in CBV.

Example: Some Principles for Call-by-Value Recursion

Using this correspondence between CBN recursion and CBV recursion, we can derive principles for CBV recursion from those for CBN recursion.

Example: Dinaturality

The dinaturality equation for CBN recursion

$$\mathsf{fix}\,(g \circ f) = g\,(\mathsf{fix}\,(f \circ g))$$

amounts to the following equation for CBV iteration

$$\mathsf{loop}\,(f \circ g) = \mathsf{loop}\,(g \circ f) \circ g$$

which corresponds to the CBV dinaturality

$$\mathsf{fix}^{\mathsf{v}}\,(G \circ (\lambda f y.F\ f\ y)) = \lambda z.G\,(\mathsf{fix}^{\mathsf{v}}\,(F \circ (\lambda g x.G\,g\,x)))\,z$$

Other examples: mutual recursion (Bekic principle) for CBV

Conclusion

Summary

Three cases of the recent investigations on recursive computation:

I Recursion from Cyclic Sharing

II Recursion and Control

III Recursion and Duality

Each of them is supported by the semantic (abstract) structures behind actual computational phenomena

---

Lesson: "Programming Languages in the Abstract"

- try to find a right level of abstraction for attacking the problem
- stick to your question, rather than the existing tools
- theoretical elegance does not contradict with practical motivations

Further Work . . .

- Part I: Good model construction techniques for traced monoidal categories needed. Related work include
  - constructions for models of asyncronomy (Selinger, MFPS'99)
  - constructions via uniformity (Hasegawa, CTCS'02)
- Part II: Interaction between recursion and general computational effects is yet to be sorted out. Related directions:
  - classifying effects (Führmann)
  - linearly used effects (Berdine et al./Hasegawa, FLOPS'02 & '04)
- Part III: Both theoretical analysis and practical applications needed.
  - theory: functional completeness (Kakutani and Hasegawa TLCA'03)
  - practice: graphical reasoning about CBN/CBV recursion (Erkök and Launchbury / Schweimeir and Jeffrey)

. . . and Speculations

Recursion, Polymorphism and Computational Effects in the Abstract

Missing: Proper denotational (abstract) semantics for
ML-Like polymorphic call-by-value languages (with/without recursion)

Related directions:

- Semantics of polymorphic computational lambda calculus
  (on-going work with Alex Simpson)
- Translation into polymorphic linear lambda calculus with recursion
  (cf. Plotkin / Pitts et al. / Ryu Hasegawa)
- "Operational semantics determines monads" approach
  (Plotkin, Power et al.)

Several issues remain to be sorted out!

Thank You