Appeared in *Proc. 3rd International Conference on Typed Lambda Calculi and Applications*. Springer LNCS1210, 1997

Recursion from Cyclic Sharing: Traced Monoidal Categories and Models of Cyclic Lambda Calculi

Masahito Hasegawa

LFCS, Department of Computer Science, University of Edinburgh
JCMB, King's Buildings, Edinburgh EH9 3JZ, Scotland
Email: mhas@dcs.ed.ac.uk

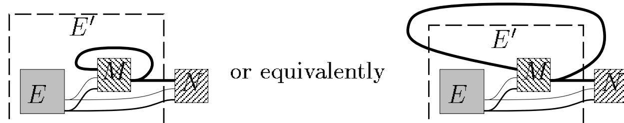
Abstract. *Cyclic sharing* (cyclic graph rewriting) has been used as a practical technique for implementing recursive computation efficiently. To capture its semantic nature, we introduce categorical models for *lambda calculi with cyclic sharing* (cyclic lambda graphs), using *notions of computation* by Moggi / Power and Robinson and *traced monoidal categories* by Joyal, Street and Verity. The former is used for representing the notion of sharing, whereas the latter for cyclic data structures. Our new models provide a semantic framework for understanding recursion created from cyclic sharing, which includes traditional models for recursion created from fixed points as special cases. Our cyclic lambda calculus serves as a uniform language for this wider range of models of recursive computation.

1 Introduction

One of the traditional methods of interpreting a recursive program in a semantic domain is to use the least fixed-point of continuous functions. However, in the real implementations of programming languages, we often use some kind of shared cyclic structure for expressing recursive environments efficiently. For instance, the following is a call-by-name operational semantics of the recursive call, in which free x may appear in M and N. We write $E \vdash M \Downarrow V$ for saying that evaluating a program M under an environment E results a value V; in a call-by-name strategy an environment assigns a free variable to a pair consisting of an environment and a program.

$$\frac{E' \vdash N \ \Downarrow\ V \quad \text{where } E' = E \cup \{x \mapsto (E', M)\}}{E \vdash \mathsf{letrec}\ x = M\ \mathsf{in}\ N \ \Downarrow\ V}$$

That is, evaluating a recursive program $\mathsf{letrec}\ x = M\ \mathsf{in}\ N$ under an environment E amounts to evaluating the subprogram N under a cyclic environment E' which references itself. One may see that it is reasonable and efficient to implement the recursive (self-referential) environment E' as a cyclic data structure as below.



Also it is known that if we implement a programming language using the technique of sharing, the use of the fixed point combinator causes some unexpected duplication of resources; it is more efficient to get recursion by cycles than by the fixed point combinator in such a setting. This fact suggests that there is a gap between the traditional approach based on fixed points and cyclically created recursion.

The aim of this paper is to introduce semantic models for understanding recursive computation created by such a cyclic data structure, especially cyclic lambda graphs as studied in [AK94]. Our task is to deal with the notion of values/non-values (which provides the notion of sharing) and the notion of cycles at the semantic level. This is done by combining Moggi's *notions of computation* [Mog88] and the notion of *traced monoidal categories* recently introduced by Joyal, Street and Verity [JSV96]. The former has been used for explaining computation and values systematically, which we apply for interpreting the notion of sharing. The latter has originally been invented for analyzing cyclic structures arising from mathematics and physics, notably knot theory (e.g. [RT90]); it is then natural to use this concept for modeling cyclic graph structure. We claim that our new models are natural objects for the study of recursive computation because they unify several aspects on recursion in just one semantic framework. The leading examples are

- the *graphical (syntactical) interpretation* of recursive programs by cyclic data structures motivated as above,
- the *domain-theoretic interpretation* in which the meaning of a recursive program $\mathsf{letrec}\ x = F(x)\ \mathsf{in}\ x$ is given by the least fixed point $\bigcup_n F^n(\bot)$, and
- the *non-deterministic interpretation* where the program $\mathsf{letrec}\ x = F(x)\ \mathsf{in}\ x$ is interpreted by $\{x \mid x = F(x)\}$, the set of all possible solutions of the equation $x = F(x)$.

Each of them has its own strong tradition in computer science. However, to our knowledge, this is the first attempt to give a uniform account on these well-known, but less-related, interpretations. Moreover, our cyclic lambda calculus serves as a uniform language for them.

Construction of this paper

We recall the definition of traced monoidal categories in Section 2. In Section 3 we observe that traces and fixed point operators are closely related in two practically interesting situations - in cartesian categories, and in a special form of monoidal adjunction known as *notions of computation*.The motivating examples above are shown to be instances of our setting. Armed with these semantic observations, in Section 4 we give the models for two simply typed lambda calculi with cyclic sharing - one with unrestricted substitution, and the other with restricted substitution on values. The two settings studied in the previous section correspond to the models of these calculi respectively; the soundness and completeness results are stated. As an application, we analyze fixed point operators definable in our calculi (Section 5).

Related work

On fixed point operators. Axiomatizations of feedback operators similar to Theorem 3.1 have been given by Bloom and Ésik in [BÉ93] where they study the dual situation (categories with coproducts). Also the same authors have considered a similar axiomatization of fixed point operators in cartesian closed categories [BÉ96]. Ignoring the difference of presentations, their "Conway cartesian categories" exactly correspond to traced cartesian categories (see the remark after Theorem 3.1). Their "Conway cartesian closed categories" are then traced cartesian closed categories with an additional condition called "abstraction identity".

On cyclic lambda calculi. Our source of cyclic lambda calculi is the version presented in [AK94]. The use of the letrec-syntax for representing cyclic sharing is not new; our presentation is inspired by a graph rewriting system in [AA95] and the call-by-need $\lambda_{\mathrm{letrec}}$-calculus in [AF96]. In this paper we concentrate on the equational characterization of the calculi; the connection between rewriting-theoretic aspects and our work remains as an important future issue. We think the relation to operational semantics should be established in this direction, especially in the connection with the call-by-need strategy [Lau93, AF96]. Also we note that our approach is applicable not only to cyclic lambda calculi but also to general cyclic graph rewriting systems.

On action calculi. The syntax and models in this paper have arisen from the study of Milner's *action calculi* [Mil96], a proposed framework for interactive computation. The use of notions of computation as models of *higher-order action calculi* [Mil94a] is developed in [GH96], whereas the relation between traced categories and *reflexive action calculi* [Mil94b] is studied by Mifsud [Mif96] and the author - axioms for reflexion are proved to be equivalent to those of trace. In fact, our cyclic lambda calculus can be seen as a fragment of a higher-order reflexive action calculus. A further study of action calculi in this paper's direction will appear in the author's forthcoming thesis (also see Example 3.10).

On Geometry of Interaction. It has been pointed out that several models of *Geometry of Interaction* [Gir89] can be regarded as traced monoidal categories (see Abramsky's survey [Abr96]). We expect that there are potential applications of our results in this direction.

2 Traced Monoidal Categories

The notion of trace we give here for symmetric monoidal categories is adopted from the original definition of traces for balanced monoidal categories [JS93] in [JSV96].

For ease of presentation, in this section we write as if our monoidal categories are strict (i.e. monoidal products are strictly associative and coherence isomorphisms are identities).

Definition 2.1. (Traced symmetric monoidal categories [JSV96])
A symmetric monoidal category $(\mathcal{T}, \otimes, I, c)$ (where c is the symmetry; $c_{X,Y} : X \otimes Y \longrightarrow Y \otimes X$) is said to be *traced* if it is equipped with a natural family of functions, called a *trace*,

$$Tr^X_{A,B} : \mathcal{T}(A \otimes X, B \otimes X) \longrightarrow \mathcal{T}(A, B)$$

subject to the following three conditions.

- **Vanishing:**

$$Tr^I_{A,B}(f) = f : A \to B$$

  where $f : A \longrightarrow B$, and

$$Tr^{X \otimes Y}_{A,B}(f) = Tr^X_{A,B}(Tr^Y_{A \otimes X, B \otimes X}(f)) : A \longrightarrow B$$

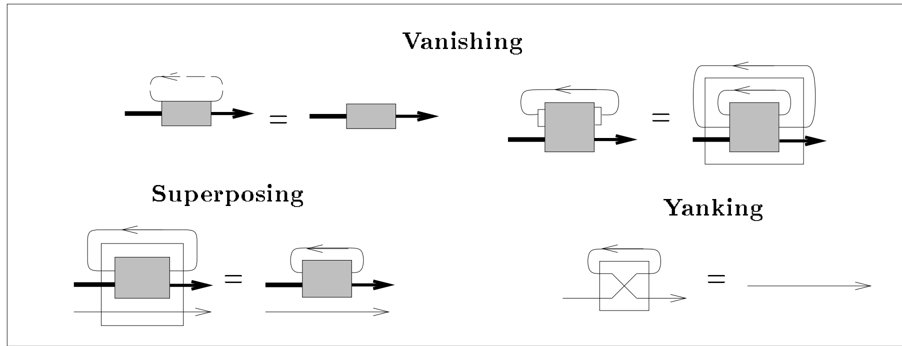
  where $f : A \otimes X \otimes Y \longrightarrow B \otimes X \otimes Y$
- **Superposing:**

$$Tr^X_{C \otimes A, C \otimes B}(id_C \otimes f) = id_C \otimes Tr^X_{A,B}(f) : C \otimes A \longrightarrow C \otimes B$$

  where $f : A \otimes X \longrightarrow B \otimes X$
- **Yanking:**

$$Tr^X_{X,X}(c_{X,X}) = id_X : X \longrightarrow X \qquad \blacksquare$$

We present the graphical version of these axioms to help with the intuition of traced categories as categories with cycles (or feedback, reflexion). Such graphical languages for various monoidal categories have been developed in [JS91].



Note that naturality of a trace can be axiomatized as follows.

- Naturality in A (**Left Tightening**)

$$Tr^X_{A,B}((g \otimes id_X); f) = g; Tr^X_{A',B}(f) : A \longrightarrow B$$

  where $f : A' \otimes X \longrightarrow B \otimes X$, $g : A \longrightarrow A'$

– Naturality in B (**Right Tightening**)

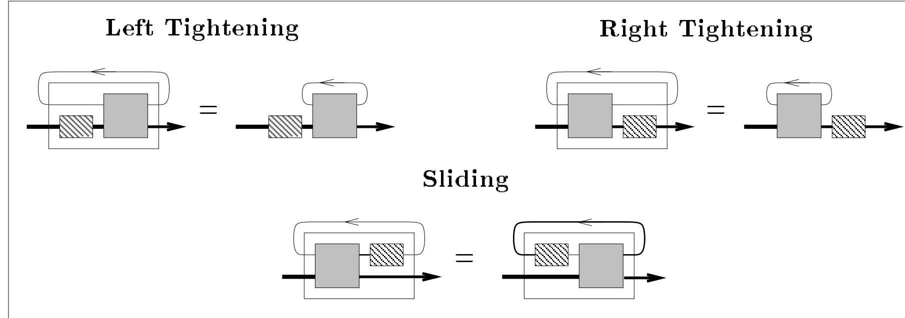
$$Tr^X_{A,B}(f;(g \otimes id_X)) = Tr^X_{A,B'}(f);g : A \longrightarrow B$$

where $f : A \otimes X \longrightarrow B' \otimes X$, $g : B' \longrightarrow B$

– Naturality in X (**Sliding**)

$$Tr^X_{A,B}(f;(id_B \otimes g)) = Tr^{X'}_{A,B}((id_A \otimes g);f) : A \longrightarrow B$$

where $f : A \otimes X \longrightarrow B \otimes X'$, $g : X' \longrightarrow X$



Remark 2.2. The axiom **Superposing** is slightly simplified from the original version in [JSV96]

$$Tr^X_{A\otimes C,B\otimes D}((id_A \otimes c_{C,X});(f \otimes g);(id_B \otimes c_{X,D})) = Tr^X_{A,B}(f) \otimes g$$

where $f : A \otimes X \longrightarrow B \otimes X$, $g : C \longrightarrow D$. Assuming axioms **Left** & **Right Tightenings**, ours is derivable from this original one, and vice versa. ■

Any compact closed category [KL80] is traced, for instance the category of sets and binary relations, and the category of finite dimensional vector spaces (see examples in next section). Moreover, the structure theorem in [JSV96] tells us that any traced symmetric monoidal category can be fully and faithfully embedded into a compact closed category (which can be obtained by a simple fraction construction). This fact, however, does not imply that the usage of traced categories is the same as that of compact closed categories. For the study of cyclic data structures, we find traced categories more useful than compact closed categories, as the latter seems to be too strong for our purpose.

3 Recursion from Traces

In this section we observe that traced categories can support recursive computation under certain circumstances. These results form the basis of our semantic analysis of "recursion created by cyclic structures" where we regard traced categories as the models of cyclic sharing.

3.1 Fixed Point Operators in Traced Cartesian Categories

Compact closed categories whose monoidal product is cartesian are trivial. This is not the case for traced categories. In fact, in [JSV96] it is shown that the category of sets and binary relations with its biproduct as the monoidal product is traced. Actually we find traced cartesian categories interesting in the context of semantics for recursive computation:

Theorem 3.1. *A cartesian category $\mathcal{C}$ is traced if and only if it has a family of functions*

$$(-)^{\dagger_{A,X}} : \mathcal{C}(A \times X, X) \longrightarrow \mathcal{C}(A, X)$$

(in below, parameters A, X may be omitted) such that

1. $(-)^{\dagger}$ *is a parametrized fixed point operator; for* $f : A \times X \longrightarrow X$, $f^{\dagger} : A \longrightarrow X$ *satisfies* $f^{\dagger} = \langle id_A, f^{\dagger}\rangle ; f$.
2. $(-)^{\dagger}$ *is natural in* A*; for* $f : A \times X \longrightarrow X$ *and* $g : B \longrightarrow A$, $((g \times id_X); f)^{\dagger} = g; f^{\dagger} : B \longrightarrow X$.
3. $(-)^{\dagger}$ *is natural in* X*; for* $f : A \times X \longrightarrow Y$ *and* $g : Y \longrightarrow X$, $(f; g)^{\dagger} = ((id_A \times g); f)^{\dagger}; g : A \longrightarrow X$.
4. $(-)^{\dagger}$ *satisfies Bekič's lemma; for* $f : A \times X \times Y \longrightarrow X$ *and* $g : A \times X \times Y \longrightarrow Y$, $\langle f, g\rangle^{\dagger} = \langle id_A, (\langle id_{A\times X}, g^{\dagger}\rangle ; f)^{\dagger}\rangle ; \langle \pi'_{A,X}, g^{\dagger}\rangle : A \longrightarrow X \times Y$.

Sketch of the proof: From a trace operator Tr, we define a fixed point operator $(-)^{\dagger}$ by

$$f^{\dagger} \;=\; Tr^X(\langle f, f\rangle) \;:\; A \longrightarrow X$$

for $f : A \times X \longrightarrow X$. Conversely, from a fixed point operator $(-)^{\dagger}$ we define a trace Tr by

$$Tr^X(f) \;=\; \langle id_A, (f; \pi'_{B,X})^{\dagger}\rangle ; f; \pi_{B,X} \;:\; A \longrightarrow B$$

(equivalently $((id_A \times \pi'_{B,X}); f)^{\dagger}; \pi_{B,X}$) for $f : A \times X \longrightarrow B \times X$. We note that these constructions are mutually inverse. □

This theorem was proved by Martin Hyland and the author independently. There are several equivalent formulations of this result. For instance, in the presence of other conditions, we can restrict 3 to the case that g is a symmetry (c.f. Lemma 1.1. of [JSV96]). For another – practically useful – example, Hyland has shown that axioms 1∼4 are equivalent to 2 and

- *(parametrized) dinaturality:* $(\langle \pi_{A,X}, f\rangle ; g)^{\dagger} = \langle id_A, (\langle \pi_{A,Y}, g\rangle ; f)^{\dagger}\rangle ; g : A \longrightarrow X$ for $f : A \times X \longrightarrow Y$ and $g : A \times Y \longrightarrow X$
- *diagonal property:* $(f^{\dagger})^{\dagger} = ((id_A \times \langle id_X, id_X\rangle); f)^{\dagger}$ for $f : A \times X \times X \longrightarrow X$.

This axiomatization is the same as that of "Conway cartesian categories" in [BÉ96]. Further variations are: 2,4 with dinaturality; and 1,2,4 with the symmetric form of 4.

Perhaps the simplest example is the opposite of the category of sets and partial functions with coproduct as the monoidal product; the trace is given by the *feedback* operator which maps a partial function $f : X \to A + X$ to $f^{\dagger} : X \to A$, determined by iterating f until we get an answer in A if exists. Such a setting is studied in [BÉ93].

An immediate consequence of Theorem 3.1 is the close relationship between traces and the least fixed point operators in traditional domain theory.

Example 3.2. (the least fixed point operator on domains)
Consider the cartesian closed category **Dom** of Scott domains and continuous functions. The least fixed point operator satisfies conditions 1∼4, thus determines a trace operator given by $Tr^X(f) = \lambda a^A.\pi(f(a, \bigsqcup^{n}(\lambda x^X.\pi'(f(a,x)))^n(\bot_X)))$ $: A \to B$ for $f : A \times X \to B \times X$. Since the least fixed point operator is the unique dinatural fixed point operator on **Dom**, the trace above is the unique one on **Dom**. ■

The same is true for several cartesian closed categories arising from domain theory. In fact, a systematic account is possible. Simpson [Sim93] has shown that, under a mild condition, in cartesian closed full subcategories of algebraic cpos, the least fixed point operator is characterized as the unique dinatural fixed point operator. On the other hand, it is easy to see that the least fixed point operators satisfy the conditions in Theorem 3.1. Therefore, in many such categories, a trace uniquely exists and is determined by the least fixed point operator. However, we note that there are at least two traces in the category of continuous lattices, an instance which does not satisfy Simpson's condition; this category has two fixed point operators which satisfy our conditions – the least one and the greatest continuous one.

Further justification of our axiomatization of fixed point operator comes from recent work on *axiomatic domain theory* which provides a more abstract and systematic treatment of domains and covers a wider range of models of domain theory than the traditional order-theoretic approach. For this, we assume some working knowledge of this topic as found in [Sim92]. Readers who are not familar with this topic may skip over to next subsection.

Example 3.3. (axiomatic domain theory)
Consider a cartesian closed category $\mathcal{C}$ (category of "predomains") equipped with a commutative monad L (the "lift") such that the Kleisli category $\mathcal{C}_L$ (category of predomains and "partial maps") is *algebraically compact* [Fre91]. This setting provides a canonical fixed point operator (derived from the *fixpoint object* [CP92]) on the category of "domains" (obtained as the co-Kleisli category of the induced comonad on the Eilenberg-Moore category $\mathcal{C}^L$) which satisfies our axioms – Bekič's lemma is proved from the algebraic compactness of $\mathcal{C}_L$ [Mog95] (this idea is due to Plotkin). Thus the requirement for solving recursive domain equations (algebraic compactness) implies that the resulting category of domains is traced. ■

Regarding these facts, we believe that traces provide a good characterization of fixed point operators in traditional denotational semantics.

We conclude this subsection by observing an attractive fact which suggests how natural our trace-fixpoint correspondence is (this is rather a digression in this paper, however). Our correspondence preserves a fundamental concept on fixed point operators called *uniformity*, also known as Plotkin's condition. This is important because fixed point operators are often canonically and uniquely characterized by this property.

Proposition 3.4. *In a traced cartesian category, the following two conditions are equivalent for any* $h : X \longrightarrow Y$.

– *(Uniformity of the trace operator) For any* f *and* g,

$$\text{if}\quad \begin{array}{ccc} A\times X & \xrightarrow{f} & B\times X \\ {\scriptstyle A\times h}\downarrow & & \downarrow{\scriptstyle B\times h} \\ A\times Y & \xrightarrow[g]{} & B\times Y \end{array} \quad \text{commutes then } Tr^X(f) = Tr^Y(g).$$

– *(Uniformity of the fixed point operator) For any* f *and* g,

$$\text{if}\quad \begin{array}{ccc} A\times X & \xrightarrow{f} & X \\ {\scriptstyle A\times h}\downarrow & & \downarrow{\scriptstyle h} \\ A\times Y & \xrightarrow[g]{} & Y \end{array} \quad \text{commutes then } f^\dagger ; h = g^\dagger. \qquad \square$$

In the case of domain-theoretic categories, the second condition is equivalent to saying that h is a strict map ($\perp$-preserving map). This fact suggests the possibility of studying the notion of strict maps and uniformity of fixed points in more general settings as in the following subsection. In particular, the first condition makes sense in any traced monoidal category.

3.2 Trace and Notions of Computation

Our observation so far says that to have an abstract trace is to have a fixed point operator in the traditional sense, provided the monoidal product is cartesian. However, regarding our motivation to model cyclic sharing, this setting is somewhat restrictive – in a cartesian category (regarded as an algebraic theory) arbitrary substitution is justified, thus there is no non-trivial notion of sharing.

To overcome this, we consider a mild generalization. Now our traced category may not be cartesian, but it is assumed to have a sub-cartesian category such that the inclusion functor preserves symmetric monoidal structure and has a right adjoint (examples will be given below). Intuitively, the sub-cartesian category is the category of "values" which can be substituted freely whereas the symmetric monoidal category part is the category of all cyclic structures which cannot be copied in general because they may contain shared resources. In this weaker setting, there still exists a fixed point operator.

Let $F : \mathcal{C} \longrightarrow \mathcal{T}$ be a faithful, identity-on-objects strict symmetric monoidal functor from a cartesian category $\mathcal{C}$ to a traced symmetric monoidal category $\mathcal{T}$, with a right adjoint. Thus we identify the objects in $\mathcal{C}$ and in $\mathcal{T}$, and F is identity as a function on objects. However, for readability, we write $A \times B$ and $A \otimes B$ for cartesian product in $\mathcal{C}$ and tensor product in $\mathcal{T}$ respectively though they are identical as F is strict symmetric monoidal. We assume a similar convention for the terminal object 1 and the unit object I.

Theorem 3.5. *Given $F : \mathcal{C} \longrightarrow \mathcal{T}$ as above, there is a family of functions*

$$(-)^{\dagger_{A,X}} : \mathcal{T}(A \otimes X, X) \longrightarrow \mathcal{T}(A, X)$$

such that

1. *$(-)^\dagger$ is a parametrized fixed point operator in the sense that, for $f : A \otimes X \longrightarrow X$ in $\mathcal{T}$, $f^\dagger : A \longrightarrow X$ satisfies $f^\dagger = \Delta_A; (id_A \otimes f^\dagger); f$ where $\Delta_A = F(\langle id_A, id_A \rangle) : A \longrightarrow A \otimes A$.*
2. *$(-)^\dagger$ is natural in A in $\mathcal{C}$; for $f : A \otimes X \longrightarrow X$ in $\mathcal{T}$ and $g : B \longrightarrow A$ in $\mathcal{C}$, $((F(g) \otimes id_X); f)^\dagger = F(g); f^\dagger : B \longrightarrow X$.*
3. *$(-)^\dagger$ is natural in X in $\mathcal{T}$; for $f : A \otimes X \longrightarrow Y$ in $\mathcal{T}$ and $g : Y \longrightarrow X$ in $\mathcal{T}$, $(f; g)^\dagger = ((id_A \otimes g); f)^\dagger; g : A \longrightarrow X$.*

Sketch of the proof: Let us write $U : \mathcal{T} \longrightarrow \mathcal{C}$ for the right adjoint of F, and $\epsilon_X : UX \longrightarrow X$ (in $\mathcal{T}$) for the counit. By definition, we have a natural isomorphism $(-)^* : \mathcal{T}(A, B) \xrightarrow{\sim} \mathcal{C}(A, UB)$. We also define $\theta_{A,X} : A \times UX \longrightarrow U(A \otimes X)$ in $\mathcal{C}$ by $\theta_{A,X} = (id_A \otimes \epsilon_X)^*$. Now we define $(-)^\dagger$ by

$$f^\dagger = Tr^{UX}(F(\theta_{A,X}; Uf); \Delta_{UX}); \epsilon_X : A \longrightarrow X \quad \text{in } \mathcal{T}$$

for $f : A \otimes X \longrightarrow X$ in $\mathcal{T}$. □

Observe that an easier construction (c.f. Theorem 3.1) $Tr^X(f; \Delta_X) : A \longrightarrow X$ from $f : A \otimes X \longrightarrow X$ in $\mathcal{T}$ does not work as a fixed point operator – the construction in Theorem 3.5 uses the adjunction in a crucial manner.

It is in general impossible to recover a trace operator from a fixed point operator which satisfies the conditions of Theorem 3.5; for instance, if $\mathcal{T}$ has a zero object 0 such that $0 \otimes A \simeq 0$ (e.g. **Rel** below), the zero map satisfies these conditions. It is an interesting question to ask if we can strengthen the conditions so that we can recover a trace operator.

A careful inspection of our construction reveals that we need the trace operator just on objects of the form UX (equivalently $F(UX)$ as F is identity-on-objects); actually it is sufficient if the full subcategory of $\mathcal{T}$ whose objects are of the form of $UX_1 \otimes \ldots \otimes UX_n$ is traced. Thus such a fixed point operator exists even in a weaker setting. It would be interesting to see if this fixed point operator determines this sub-trace structure. It would be more interesting to see if there is a good connection between such a fixed point operator and fixed point operators in models of intuitionistic linear logic as studied in [Bra95].

An observation corresponding to Proposition 3.4 is as follows: for any h in $\mathcal{T}$, if $F(U(h))$ satisfies the uniformity condition for the trace operator then h satisfies the uniformity condition for the fixed point operator.

Note that our setting is equivalent to saying that we have a cartesian category $\mathcal{C}$ with a monad $U \circ F$ on it, which satisfies the mono-requirement and has a commutative tensorial strength θ, such that the Kleisli category $\mathcal{T}$ is traced. In other words, we are dealing with some *notions of computation* in the sense of Moggi [Mog88] with extra structure (trace). Our definition is inspired by a recent reformulation of notions of computation by Power and Robinson [PR96].

Definition 3.6. A *traced computational model* is a faithful, identity-on-object strict symmetric monoidal functor $F : \mathcal{C} \longrightarrow \mathcal{T}$ where $\mathcal{C}$ is a cartesian category and $\mathcal{T}$ a traced symmetric monoidal category, such that the functor $F(-) \otimes X : \mathcal{C} \longrightarrow \mathcal{T}$ has a right adjoint $X \Rightarrow (-) : \mathcal{T} \longrightarrow \mathcal{C}$: thus $\mathcal{T}(F(-) \otimes X, -) \simeq \mathcal{C}(-, X \Rightarrow -)$. ■

$X \Rightarrow Y$ is the so-called *Kleisli exponent*; if $\mathcal{C}$ is cartesian closed, $X \Rightarrow Y$ is obtained as $(UY)^X$. As a traced computational model satisfies the assumption in Theorem 3.5 (a right adjoint of F is given by $I \Rightarrow (-)$), there is a fixed point operator in its traced category. The right adjoint $X \Rightarrow (-)$ can be used to interpret higher-order (higher-type) computation. Thus traced computational models have enough structure to interpret higher-order recursive computation; later we see how they can be used as the models of a simply typed lambda calculus with cyclic sharing.

To help with the intuition, we shall give a selection of traced computational models below. Most of them have already been mentioned in Section 1.

Example 3.7. (traced cartesian closed categories)
A traced cartesian closed category is a traced computational model in which the cartesian category part and the traced category part are identical. Examples include many domain-theoretic categories such as Example 3.2. ■

Example 3.8. (non-deterministic model)
The inclusion from the category **Set** of sets and functions to the category **Rel** of sets and binary relations (with the direct product of sets as the symmetric monoidal product) forms a traced computational model: $\mathbf{Rel}(A \otimes X, B) \simeq \mathbf{Set}(A, \mathbf{Rel}(X, B))$. The trace operator on **Rel**, induced by the compact closed structure of **Rel**, is given as follows: for a relation $R : A \otimes X \longrightarrow B \otimes X$, we define a relation $Tr^X(R) : A \longrightarrow B$ by $(a, b) \in Tr^X(R)$ iff $((a, x), (b, x)) \in R$ for an $x \in X$ (here a relation from A to B is given as a subobject of $A \times B$). The parametrized fixed point operator $(-)^\dagger$ on **Rel** is given by

$$R^\dagger \;=\; \{(a, x) \mid \exists S \subseteq X \; S = \{y \mid \exists x \in S \; ((a, x), y) \in R\} \;\&\; x \in S\} \;:\; A \longrightarrow X$$

for $R : A \otimes X \longrightarrow X$ (and $R^\dagger$ is not the zero map!). Note that we can use an elementary topos instead of **Set**, which may provide a computationally more sophisticated model. ■

Example 3.9. (finite dimensional vector spaces over a finite field)
Let F_2 be the field with just two elements (thus its characteristic is 2), and $\mathbf{Vect}^{\mathrm{fin}}_{F_2}$ be the category of finite dimensional vector spaces (with chosen bases) over F_2. There is a strict symmetric monoidal functor from the category of finite sets to $\mathbf{Vect}^{\mathrm{fin}}_{F_2}$ which maps a set S to a vector space with the basis S, and this functor has a right adjoint (the underlying functor). Since $\mathbf{Vect}^{\mathrm{fin}}_{F_2}$ is traced (in the very classical sense), this is an instance of traced computational models. Note that this example is similar to the previous one – compare the matrix representation of binary relations and that of linear maps. ■

Example 3.10. (higher-order reflexive action calculi)
Recent work [GH96, Mif96] on action calculi [Mil96] shows that the higher-order reflexive extension of an action calculus [Mil94a, Mil94b] forms a traced computational model. In this calculus the fixed point operator $(-)^{\dagger}$ is given by

$$t^{\dagger} \;=\; \uparrow_{\epsilon\Rightarrow n} ((x^{\epsilon\Rightarrow n})\ulcorner(\mathbf{id}_m \otimes \langle x\rangle \cdot \mathbf{ap}_{\epsilon,n}) \cdot t\urcorner \cdot \mathbf{copy}_{\epsilon\Rightarrow n}) \cdot \mathbf{ap}_{\epsilon,n} \;:\; m \longrightarrow n$$

for $t : m \otimes n \longrightarrow n$. Mifsud gives essentially the same operator $\mathbf{iter}_f(t)$ in his thesis [Mif96]. Using this, we can present recursion operators in various process calculi, typically the replication operator. ■

4 Semantics of Lambda Calculi with Cyclic Sharing

We introduce two simply typed lambda calculi enriched with the notion of cyclic sharing, the *simply typed* $\lambda_{\mathrm{letrec}}$*-calculus* and $\lambda^{v}_{\mathrm{letrec}}$*-calculus* in which cyclically shared resources are represented in terms of the letrec syntax. It is shown that traced cartesian closed categories and traced computational models are sound and complete models of these calculi respectively.

4.1 The Syntax and Axioms

As the semantic observation we have seen suggests, the simply typed $\lambda^{v}_{\mathrm{letrec}}$-calculus is designed as a modification of Moggi's computational lambda calculus [Mog88]; we replace the let-syntax by the letrec-syntax which allows cyclic bindings.

In this section, we fix a set of *base types.*

Types

$$\sigma, \tau \ldots \;::=\; b \mid \sigma \Rightarrow \tau \quad \text{(where } b \text{ is a base type)}$$

Syntactic Domains

Variables	$x, y, z \ldots$
Raw Terms	$M, N \ldots \;::=\; x \mid \lambda x.M \mid MN \mid \mathsf{letrec}\ D\ \mathsf{in}\ N$
Values	$V, W \ldots \;::=\; x \mid \lambda x.M$
Declarations	$D \ldots \;::=\; x = M \mid x = M, D$

In a declaration, binding variables are assumed to be disjoint.

Typing

$$\frac{}{\Gamma, x:\sigma \vdash x:\sigma}\ \text{Variable} \qquad \frac{\Gamma, x:\sigma, y:\sigma', \Gamma' \vdash M:\tau}{\Gamma, y:\sigma', x:\sigma, \Gamma' \vdash M:\tau}\ \text{Exchange}$$

$$\frac{\Gamma, x:\sigma \vdash M:\tau}{\Gamma \vdash \lambda x.M : \sigma \Rightarrow \tau}\ \text{Abstraction} \qquad \frac{\Gamma \vdash M:\sigma \Rightarrow \tau \quad \Gamma \vdash N:\sigma}{\Gamma \vdash MN:\tau}\ \text{Application}$$

$$\frac{\Gamma, x_1:\sigma_1,\ldots,x_n:\sigma_n \vdash M_i:\sigma_i\ (i=1,\ldots,n) \quad \Gamma, x_1:\sigma_1,\ldots,x_n:\sigma_n \vdash N:\tau}{\Gamma \vdash \mathsf{letrec}\ x_1 = M_1,\ldots,x_n = M_n\ \mathsf{in}\ N:\tau}\ \mathsf{letrec}$$

Axioms

Identity	$\mathsf{letrec}\ x = M\ \mathsf{in}\ x$	$= M \quad (x \notin FV(M))$
Associativity	$\mathsf{letrec}\ y = (\mathsf{letrec}\ D_1\ \mathsf{in}\ M), D_2\ \mathsf{in}\ N$	$= \mathsf{letrec}\ D_1, y = M, D_2\ \mathsf{in}\ N$
	$\mathsf{letrec}\ D_1\ \mathsf{in}\ \mathsf{letrec}\ D_2\ \mathsf{in}\ M$	$= \mathsf{letrec}\ D_1, D_2\ \mathsf{in}\ M$
Permutation	$\mathsf{letrec}\ D_1, D_2, D\ \mathsf{in}\ N$	$= \mathsf{letrec}\ D_2, D_1, D\ \mathsf{in}\ N$
Commutativity	$(\mathsf{letrec}\ D\ \mathsf{in}\ M)N$	$= \mathsf{letrec}\ D\ \mathsf{in}\ MN$
	$M(\mathsf{letrec}\ D\ \mathsf{in}\ N)$	$= \mathsf{letrec}\ D\ \mathsf{in}\ MN$
β	$(\lambda x.M)N$	$= \mathsf{letrec}\ x = N\ \mathsf{in}\ M$
σ_v	$\mathsf{letrec}\ x = V, D[x]\ \mathsf{in}\ M$	$= \mathsf{letrec}\ x = V, D[V]\ \mathsf{in}\ M$
	$\mathsf{letrec}\ x = V, D\ \mathsf{in}\ M[x]$	$= \mathsf{letrec}\ x = V, D\ \mathsf{in}\ M[V]$
	$\mathsf{letrec}\ x = V\ \mathsf{in}\ M$	$= M \quad (x \notin FV(V) \cup FV(M))$
η_0	$\lambda x.yx$	$= y$

Both sides of equations must have the same type under the same typing context; we will work just on well-typed terms. We assume the usual conventions on variables.

We remark that axioms Identity, Associativity, Permutation and Commutativity ensure that two $\lambda^v_{\mathrm{letrec}}$-terms are identified if they correspond to the same cyclic directed graph; thus they are a sort of structural congruence, rather than representing actual computation. β creates a sharing from a function application. σ_v describes the substitution of values (the first two for the dereference, the last one for the garbage collection). In $M[x]$ and $D[x]$, $[x]$ denotes a free occurrence of x. From β, σ_v and η_0, we have the "call-by-value" $\beta\eta$-equations:

Lemma 4.1. *In $\lambda^v_{\mathrm{letrec}}$-calculus, the following are derivable.*

$$\begin{array}{ll} \beta_v & (\lambda x.M)V = M\{V/x\} \\ \eta_v & (\lambda x.Vx) \ = V \quad (x \notin FV(V)) \quad \Box \end{array}$$

We think it is misleading to relate this calculus to the call-by-value operational semantics; restricting substitutions on values does not mean that this calculus is for call-by-value. Rather, our equational theory is fairly close to the *call-by-need* calculus proposed in [AF96], which corresponds to a version of lazy implementations of the call-by-name operational semantics. We expect that this connection is the right direction to relate our calculus to an operational semantics.

Also we define a "strengthened" version in which arbitrary substitution and η-reduction are allowed (thus any term is a value):

$$\begin{array}{lll}
\sigma & \mathsf{letrec}\ x = N, D[x]\ \mathsf{in}\ M = \mathsf{letrec}\ x = N, D[N]\ \mathsf{in}\, M \\
& \mathsf{letrec}\ x = N, D\ \mathsf{in}\ M[x] = \mathsf{letrec}\ x = N, D\ \mathsf{in}\, M[N] \\
& \mathsf{letrec}\ x = N\ \mathsf{in}\ M \qquad = M \quad (x \notin FV(M)) \\
\eta & \lambda x.Mx \qquad\qquad\qquad\ = M \quad (x \notin FV(M))
\end{array}$$

We shall call this version the *simply typed* $\lambda_{\mathrm{letrec}}$*-calculus* – this corresponds to the calculus in [AK94] ignoring the typing and the extensionality (η-axiom).

4.2 Interpretation into Traced Computational Models

We just present the case of the $\lambda^{v}_{\mathrm{letrec}}$-calculus; the case of the $\lambda_{\mathrm{letrec}}$-calculus is obtained just by replacing a traced computational model by a traced cartesian closed category.

Let us fix a traced computational model $F : \mathcal{C} \longrightarrow \mathcal{T}$, and choose an object $[\![b]\!]$ for each base type b. The interpretation of arrow types is then defined by $[\![\sigma \Rightarrow \tau]\!] = [\![\sigma]\!] \Rightarrow [\![\tau]\!]$. We interpret a $\lambda^{v}_{\mathrm{letrec}}$-term (with its typing environment) $x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M : \tau$ to an arrow $[\![x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash M : \tau]\!] : [\![\sigma_1]\!] \otimes \ldots \otimes [\![\sigma_n]\!] \longrightarrow [\![\tau]\!]$ in $\mathcal{T}$ as follows.

$$\begin{array}{ll}
[\![x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash x_i : \sigma_i]\!] & = F(\pi_i) \ \text{ where } \pi_i \text{ is the } i\text{-th projection} \\
[\![\Gamma \vdash \lambda x.M : \sigma \Rightarrow \tau]\!] & = F(\mathbf{cur}([\![\Gamma, x : \sigma \vdash M : \tau]\!])) \\
[\![\Gamma \vdash M^{\sigma \Rightarrow \tau} N^{\sigma} : \tau]\!] & = \Delta; ([\![\Gamma \vdash M : \sigma \Rightarrow \tau]\!] \otimes [\![\Gamma \vdash N : \tau]\!]); \mathbf{ap} \\
[\![\Gamma \vdash \mathsf{letrec}\ x_1 = M_1^{\sigma_1}, .., x_k = M_k^{\sigma_k}\ \mathsf{in}\ N : \tau]\!] = & \\
\quad \Delta; (id \otimes Tr^{[\![\sigma_1]\!] \otimes .. \otimes [\![\sigma_k]\!]}(\Delta_k; ([\![\Gamma' \vdash M_1 : \sigma_1]\!] \otimes \ldots \otimes [\![\Gamma' \vdash M_k : \sigma_k]\!]); \Delta)); [\![\Gamma' \vdash N : \tau]\!] &
\end{array}$$

where $\mathbf{ap}_{A,B} : (A \Rightarrow B) \otimes A \longrightarrow B$ is the counit of the adjoint $F(-) \otimes A \dashv A \Rightarrow (-)$, and $\mathbf{cur} : \mathcal{T}(FA \otimes B, C) \longrightarrow \mathcal{C}(A, B \Rightarrow C)$ the associated natural bijection. In the last case, Γ' is $\Gamma, x_1 : \sigma_1, \ldots, x_k : \sigma_k$ and Δ_k is the k-times copy ($\Delta_{kA} = F(\langle \underbrace{id, \ldots, id}_{k \text{ times}} \rangle) : A \longrightarrow \underbrace{A \otimes \ldots \otimes A}_{k \text{ times}}$). Note that values are first interpreted in $\mathcal{C}$ (following Moggi's account, $\mathcal{C}$ is the category of values) and then lifted to $\mathcal{T}$ via F.

A straightforward calculation shows that traced computational models are sound for the $\lambda^{v}_{\mathrm{letrec}}$-calculus (and the same for traced cartesian closed categories and the $\lambda_{\mathrm{letrec}}$-calculus):

Theorem 4.2. *(Soundness)*

- *For any traced computational model with chosen object* $[\![b]\!]$ *for each base type* b*, this interpretation is sound; if* $\Gamma \vdash M : \sigma$*,* $\Gamma \vdash N : \sigma$ *and* $M = N$ *in the* $\lambda^{v}_{\mathrm{letrec}}$*-calculus then* $[\![\Gamma \vdash M : \sigma]\!] = [\![\Gamma \vdash N : \sigma]\!]$*.*
- *For any traced cartesian closed category with chosen object* $[\![b]\!]$ *for each base type* b*, this interpretation is sound; if* $\Gamma \vdash M : \sigma$*,* $\Gamma \vdash N : \sigma$ *and* $M = N$ *in the* $\lambda_{\mathrm{letrec}}$*-calculus then* $[\![\Gamma \vdash M : \sigma]\!] = [\![\Gamma \vdash N : \sigma]\!]$*.* □

Example 4.3. (domain-theoretic model)
As we already noted, **Dom** is a traced cartesian closed category (hence also a traced computational model). The interpretation of a λ_{letrec}-term $\vdash \mathsf{letrec}\ x = M\ \mathsf{in}\ x : \sigma$ in **Dom** is just the least fixed point $\bigcup_n F^n(\bot)$ where $F : [\![\sigma]\!] \longrightarrow [\![\sigma]\!]$ is the interpretation of $x : \sigma \vdash M : \sigma$. ■

Example 4.4. (non-deterministic model)
In **Rel** (Example 3.8), a $\lambda^v_{\text{letrec}}$-term is interpreted as the set of "all possible solutions of the recursive equation". The interpretation of $\vdash \mathsf{letrec}\ x = M\ \mathsf{in}\ x : \sigma$ is just the set $\{x \in [\![\sigma]\!] \mid (x, x) \in [\![x : \sigma \vdash M : \sigma]\!]\}$ (a subobject of $[\![\sigma]\!] = 1 \times [\![\sigma]\!]$). For instance,

$$
\begin{array}{lll}
[\![\vdash \mathsf{letrec}\ x = x\ \mathsf{in}\ x : \sigma]\!] & = [\![\sigma]\!] & : 1 \longrightarrow [\![\sigma]\!] \\
[\![\vdash \mathsf{letrec}\ x = x^2\ \mathsf{in}\ x : \mathrm{nat}]\!] & = \{0, 1\} & : 1 \longrightarrow \mathbf{N} \\
[\![\vdash \mathsf{letrec}\ x = x + 1\ \mathsf{in}\ x : \mathrm{nat}]\!] & = \emptyset & : 1 \longrightarrow \mathbf{N}
\end{array}
$$

(for the latter two cases we enrich the calculus with natural numbers). Note that this model is sound for the $\lambda^v_{\text{letrec}}$-calculus, but not for the λ_{letrec}-calculus – since we cannot copy non-deterministic computation, this model is "resource-sensitive". ■

Moreover, we can construct a term model (enriched with the unit and product types) to which the $\lambda^v_{\text{letrec}}$-calculus (or λ_{letrec}-calculus) is faithfully interpreted. Actually it is possible to show that the $\lambda^v_{\text{letrec}}$-calculus is faithfully embedded into the higher-order reflexive action calculus (Example 3.10) which is an instance of traced computational models. Thus we also have completeness:

Theorem 4.5. *(Completeness)*

- *If* $[\![\Gamma \vdash M : \sigma]\!] = [\![\Gamma \vdash N : \sigma]\!]$ *for every traced computational model, then* $M = N$ *in the* $\lambda^v_{\text{letrec}}$*-calculus.*
- *If* $[\![\Gamma \vdash M : \sigma]\!] = [\![\Gamma \vdash N : \sigma]\!]$ *for every traced cartesian closed category, then* $M = N$ *in the* λ_{letrec}*-calculus.* □

Remark 4.6. To represent the parametrized fixed point operator given in Theorem 3.5 we have to extend the $\lambda^v_{\text{letrec}}$-calculus with a *unit type* unit which has a unique value $*$:

$$
\frac{}{\Gamma \vdash * : \mathsf{unit}}\ \text{Unit} \qquad V \ = \ * \quad (V : \mathsf{unit})
$$

The interpretation of the unit type in a traced computational model is just the terminal object (unit object). The type constructor $\mathsf{unit} \Rightarrow (-)$ then plays the role of the right adjoint of the inclusion from the category of values to the category of terms. We define the parametrized fixed point operator by

$$
\frac{\Gamma, x : \sigma \vdash M : \sigma}{\Gamma \vdash \mu x^\sigma.M \ \equiv \ \mathsf{letrec}\ f^{\mathsf{unit} \Rightarrow \sigma} = \lambda y^{\mathsf{unit}}.((\lambda x^\sigma.M)(f*))\ \mathsf{in}\ f* \ : \ \sigma}
$$

which satisfies $\mu x.M = (\lambda x.M)(\mu x.M)$, but may not satisfy $\mu x.M = M\{\mu x.M/x\}$ in the $\lambda^v_{\text{letrec}}$-calculus because $\mu x.M$ may not be a value in general. The operator Y_3 in the next section is essentially same as this fixed point operator, except for avoiding to use unit. ■

We could give the untyped version and its semantic models – by a reflexive object in a traced computational model (or a traced cartesian closed category). Regarding the results in Section 3, we can establish the connection between the dinatural diagonal fixed point operator in a model of the untyped λ_{letrec}-calculus and the trace operator of the cartesian closed category. It would be interesting to compare recursion created by untypedness and recursion created by trace (cyclic sharing) in such models.

5 Analyzing Fixed Points

In the $\lambda^v_{\text{letrec}}$-calculus, several (weak) fixed point operators are definable – this is not surprising, because there are several known encodings of fixed point operators in terms of cyclic sharing. However, it is difficult to see that they are not identified by our equational theory – syntactic reasoning for cyclic graph structures is not an easy task, as the non-confluency result in [AK94] suggests. On the other hand, in many traditional models for recursive computation, all of them have the same denotational meaning mainly because we cannot distinguish values from non-values in such models.

One purpose of developing the traced computational models is to give a clear semantic account for these several recursive computations created from cyclic sharing. Though this topic has not yet been fully developed, we shall give some elementary analysis using the $\lambda^v_{\text{letrec}}$-calculus and a traced computational model (**Rel**).

We define $\lambda^v_{\text{letrec}}$-terms $\Gamma \vdash Y_i(M) : \sigma$ $(i = 1, 2, 3)$ for given term $\Gamma \vdash M : \sigma \Rightarrow \sigma$ as follows.

$$\begin{aligned}
Y_1 &= \mathsf{letrec}\ \mathrm{fix}^{(\sigma\Rightarrow\sigma)\Rightarrow\sigma} = \lambda f^{\sigma\Rightarrow\sigma}.f(\mathrm{fix}\ f)\ \mathsf{in}\ \mathrm{fix} \\
Y_2 &= \lambda f^{\sigma\Rightarrow\sigma}.\mathsf{letrec}\ x^{\sigma} = fx\ \mathsf{in}\ x \\
Y_3(M) &= \mathsf{letrec}\ g^{\tau\Rightarrow\sigma} = \lambda y^{\tau}.M(gy)\ \mathsf{in}\ gN \\
&\quad (N \text{ is a closed term of type } \tau, \text{ e.g. } \mathsf{letrec}\ x = x\ \mathsf{in}\ x\ : \tau)
\end{aligned}$$

Each of them can be used as a fixed point operator, but their behaviours are not the same. For instance, it is known that Y_2 is more efficient than others, under the call-by-need evaluation strategy [Lau93]. Y_1 satisfies the fixed point equation $YV = V(YV)$ for any value $V : \sigma \Rightarrow \sigma$.

$$\begin{aligned}
Y_1 M &= \mathsf{letrec}\ \mathrm{fix} = \lambda f.f(\mathrm{fix}\ f)\ \mathsf{in}\ \mathrm{fix} M && \text{Commutativity} \\
&= \mathsf{letrec}\ \mathrm{fix} = \lambda f.f(\mathrm{fix}\ f)\ \mathsf{in}\ (\lambda f.f(\mathrm{fix}\ f))M && \sigma_v \\
&= \mathsf{letrec}\ \mathrm{fix} = \lambda f.f(\mathrm{fix}\ f)\ \mathsf{in}\ \mathsf{letrec}\ f' = M\ \mathsf{in}\ f'(\mathrm{fix}\ f') && \beta \\
&= \mathsf{letrec}\ f' = M\ \mathsf{in}\ \mathsf{letrec}\ \mathrm{fix} = \lambda f.f(\mathrm{fix}\ f)\ \mathsf{in}\ f'(\mathrm{fix}\ f') && \text{Associativity, Permutation} \\
&= \mathsf{letrec}\ f' = M\ \mathsf{in}\ f'((\mathsf{letrec}\ \mathrm{fix} = \lambda f.f(\mathrm{fix}\ f)\ \mathsf{in}\ \mathrm{fix})f') && \text{Commutativity} \\
&= \mathsf{letrec}\ f' = M\ \mathsf{in}\ f'(Y_1 f') \\
&(= M(Y_1 M) \quad \text{if } M \text{ is a value})
\end{aligned}$$

Y_2 satisfies $Y_2 M = M(Y_2 M)$ only when Mx is equal to a value (hence M is

supposed to be a higher-order value). If $M = \lambda y.V$ for some value V,

$$\begin{array}{rll} Y_2 M & = \mathsf{letrec}\ x = (\lambda y.V)x\ \mathsf{in}\ x & \beta_v \\ & = \mathsf{letrec}\ x = V\{x/y\}\ \mathsf{in}\ x & \beta_v \\ & = \mathsf{letrec}\ x = V\{x/y\}\ \mathsf{in}\ V\{x/y\} & \sigma_v \\ & = \mathsf{letrec}\ x = (\lambda y.V)x\ \mathsf{in}\ (\lambda y.V)x & \beta_v \\ & = (\lambda y.V)(\mathsf{letrec}\ x = (\lambda y.V)x\ \mathsf{in}\ x) & \text{Commutativity} \\ & = M(Y_2 M) & \end{array}$$

Y_3 satisfies $Y_3(M) = M(Y_3(M))$ for any term $M : \sigma \Rightarrow \sigma$ (thus is a "true" fixed point operator).

$$\begin{array}{rll} Y_3(M) & = \mathsf{letrec}\ g = \lambda y.M(gy)\ \mathsf{in}\ (\lambda y.M(gy))N & \sigma_v \\ & = \mathsf{letrec}\ g = \lambda y.M(gy)\ \mathsf{in}\ \mathsf{letrec}\ y' = N\ \mathsf{in}\ M(gy') & \beta \\ & = \mathsf{letrec}\ g = \lambda y.M(gy)\ \mathsf{in}\ M(g(\mathsf{letrec}\ y' = N\ \mathsf{in}\ y')) & \text{Commutativity} \\ & = M(\mathsf{letrec}\ g = \lambda y.M(gy)\ \mathsf{in}\ g(\mathsf{letrec}\ y' = N\ \mathsf{in}\ y')) & \text{Commutativity} \\ & = M(\mathsf{letrec}\ g = \lambda y.M(gy)\ \mathsf{in}\ gN) & \text{Identity} \\ & = M(Y_3(M)) & \end{array}$$

The interpretation of these operators in a traced computational model is as follows.

$$\begin{array}{ll} [\![\vdash Y_1]\!] & = Tr^{(A \Rightarrow A) \Rightarrow A}(F(\mathbf{cur}((id \otimes \Delta); (\mathbf{ap} \otimes id); c; \mathbf{ap})); \Delta) \\ [\![\vdash Y_2]\!] & = F(\mathbf{cur}(Tr^A(\mathbf{ap}; \Delta))) \\ [\![\Gamma \vdash Y_3(M)]\!] & = (Tr^{B \Rightarrow A}(F(\mathbf{cur}(([\![\Gamma \vdash M : \sigma \Rightarrow \sigma]\!] \otimes \mathbf{ap}); \mathbf{ap})); \Delta) \otimes [\![\vdash N : \tau]\!]); \mathbf{ap} \end{array}$$

where $A = [\![\sigma]\!]$ and $B = [\![\tau]\!]$. They have the different interpretations in $\mathbf{Rel}$, hence are not identified in the $\lambda^v_{\mathrm{letrec}}$-calculus. Assume that $S = [\![\vdash M : \sigma \Rightarrow \sigma]\!] \subseteq \mathbf{Rel}(A, A)$. Then

$$[\![\vdash Y_1(M) : \sigma]\!] = \bigcup_{f \in S} \bigcup_{(A';f) = A' \subseteq A} A' \qquad [\![\vdash Y_2(M) : \sigma]\!] = \bigcup_{f \in S} \{x \mid (x, x) \in f\}$$

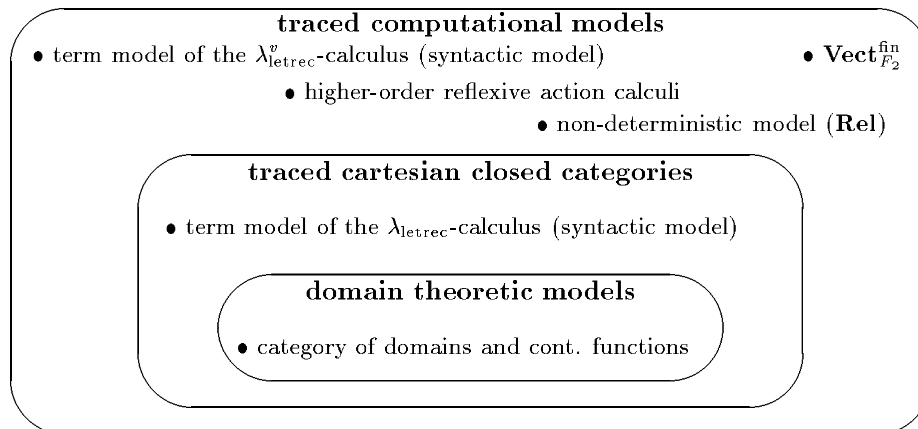
whereas

$$[\![\vdash Y_3(M) : \sigma]\!] = \bigcup_{(A';\bigcup S) = A' \subseteq A} A'$$

(In the definition of Y_3, we take $N : \tau$ as $\mathsf{letrec}\ x = x\ \mathsf{in}\ x : \tau$.)

6 Conclusion

We have presented new semantic models for interpreting cyclic sharing in terms of traced monoidal categories and notions of computation, and shown the connections with cyclic lambda calculi and with traditional semantics for recursive computation. We have also demonstrated that our framework covers a wider range of models of recursion than the traditional approach. We summarize this situation, together with examples in this paper, in the diagram below.



Acknowledgements

I am deeply grateful to Martin Hyland and John Power for helpful discussions, suggestions and encouragement. I also thank Philippa Gardner, Alex Mifsud, Marcelo Fiore, Alex Simpson and Gordon Plotkin for their comments and encouragement.

References

Abr96. S. Abramsky, Retracing some paths in process algebra. In *Proc. 7th Int. Conf. Concurrency Theory (CONCUR'96)*, Springer LNCS 1119, pages 1-17, 1996.

AA95. Z. Ariola and Arvind, Properties of a first-order functional language with sharing. *Theoretical Computer Science* 146, pages 69-108, 1995.

AF96. Z. Ariola and M. Felleisen, A call-by-need lambda calculus. Technical report CIS-TR-96-97, 1996. To appear in *Journal of Functional Programming.*

AK94. Z. Ariola and J. Klop, Cyclic lambda graph rewriting. In *Proc. 9th Symposium on Logic in Computer Science (LICS'94)*, pages 416-425, 1994.

BÉ93. S. L. Bloom and Z. Ésik, *Iteration Theories.* EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1993.

BÉ96. S. L. Bloom and Z. Ésik, Fixed point operators on ccc's. Part I. *Theoretical Computer Science* 155, pages 1-38, 1996.

Bra95. T. Braüner, The Girard translation extended with recursion. In *Proc. Computer Science Logic 1994 (CSL'94)*, Springer LNCS 933, pages 31-45, 1995.

CP92. R. L. Crole and A. M. Pitts, New foundations for fixpoint computations: Fix hyperdoctrines and the fix logic. *Information and Computation* 98, pages 171-210, 1992.

Fre91. P. Freyd, Algebraically complete categories. In *Proc. 1990 Como Category Theory Conference*, Springer LNM 1144, pages 95-104, 1991.

GH96. P. Gardner and M. Hasegawa, On higher-order action calculi and notions of computation. Draft, LFCS, University of Edinburgh, 1996.

Gir89. J. -Y. Girard, Geometry of interaction I: interpretation of system F. In *Logic Colloquium '88*, pages 221-260, North-Holland, 1989.

JS91. A. Joyal and R. Street, The geometry of tensor calculus I. *Advances in Mathematics* 88, pages 55-113, 1991.

JS93. A. Joyal and R. Street, Braided tensor categories. *Advances in Mathematics* 102, pages 20-78, 1993.

JSV96. A. Joyal, R. Street and D. Verity, Traced monoidal categories. *Mathematical Proceedings of the Cambridge Philosophical Society* 119(3), pages 447-468, 1996.

KL80. M. Kelly and M. L. Laplaza, Coherence for compact closed categories. *Journal of Pure and Applied Algebra* 19, pages 193-213, 1980.

Lau93. J. Launchbury, A natural semantics for lazy evaluation. In *Proc. 21st ACM Symp. Principles of Programming Languages (POPL'93)*, pages 144-154, 1993.

Mif96. A. Mifsud, *Control structures.* PhD thesis, LFCS, University of Edinburgh, 1996.

Mil94a. R. Milner, Higher-order action calculi. In *Proc. Computer Science Logic 1992 (CSL'92)*, Springer LNCS 832, pages 238-260, 1994.

Mil94b. R. Milner, Action calculi V: reflexive molecular forms (with Appendix by O. Jensen). Third draft, July 1994.

Mil96. R. Milner, Calculi for interaction. *Acta Informatica* 33(8), pages 707-737, 1996.

Mog88. E. Moggi, Computational lambda-calculus and monads. Technical report ECS-LFCS-88-66, LFCS, University of Edinburgh, 1988.

Mog95. E. Moggi, Metalanguages and applications. Draft, 1995.

PR96. A. J. Power and E. P. Robinson, Premonoidal categories and notions of computation. 1996. To appear in *Mathematical Structures in Computer Science.*

RT90. N. Yu. Reshetikhin and V. G. Turaev, Ribbon graphs and their invariants derived from quantum groups. *Communications in Mathematical Physics* 127, pages 1-26, 1990.

Sim92. A. Simpson, Recursive types in Kleisli categories. Manuscript, LFCS, University of Edinburgh, 1992.

Sim93. A. Simpson, A characterisation of the least-fixed-point operator by dinaturality. *Theoretical Computer Science* 118, pages 301-314, 1993.