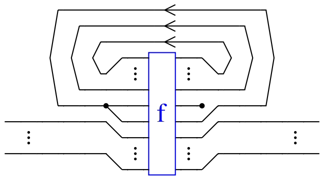
Premonoidal categories and a graphical view of programs

Alan Jeffrey
School of Cognitive and Computing Sciences
University of Sussex
Brighton BN1 9QH
UK
alanje@cogs.susx.ac.uk

December 1997



Abstract

This paper describes the relationship between two different presentations of the semantics of programs:

- *Mixed data and control flow graphs* are commonly used in software engineering as a semi-formal notation for describing and analysing algorithms.
- *Category theory* is used as an abstract presentation of the mathematical structures used to give a formal semantics to programs.

In this paper, we formalize an appropriate notion of flow graph, and show that acyclic flow graphs form the initial *symmetric premonoidal category*. Thus, giving a semantics for a programming language in flow graphs uniquely determines a semantics in *any* symmetric premonoidal category.

For languages with recursive definitions, we show that cyclic flow graphs form the initial *partially traced cartesian category*.

Finally, we conclude with some more speculative work, showing how closed structure (to represent higher-order functions) or two-categorical structure (to represent operational semantics) might be included in this graphical framework.

The semantics has been implemented as a Java applet, which takes a program text and draws the corresponding flow graph (all the diagrams in this paper are drawn using this applet).

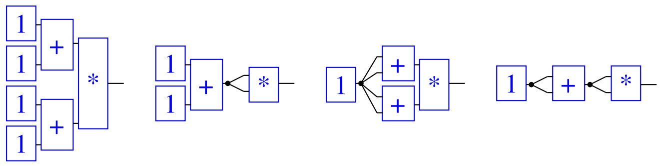
The categorical presentation is based on Power and Robinson's *premonoidal categories* and Joyal, Street and Verity's *monoidal traced categories*, and uses similar techniques to Hasegawa's semantics for recursive declarations. The closed and two-categorical structure is related to Gardner's name-free presentation of Milner's action calculi.

1 Introduction

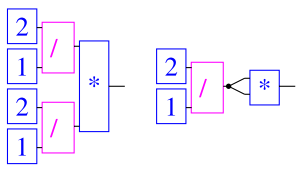
Two techniques for giving semantics of programs are *graphically* using data flow and control flow diagrams, or *categorically* using mathematical structures such as monads or premonoidal categories. Usually, the graphical presentation is semi-formal, and forms part of the dataflow-oriented design process taught to most computer science undergraduates, whereas the categorical presentation is used in giving formal semantics, and is the subject of specialist research.

In this paper, we shall give a formalization of flow graphs, and show how this can be used to give a categorical semantics in a framework based on Power and Robinson's premonoidal categories and Joyal, Street and Verity's traced monoidal categories.

As an example of the flow graphs described in this paper, consider a nondeterministic programming language with a single imperative integer variable. Such a programming language contains expressions which can be drawn as data flow diagrams such as:

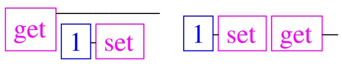


We can also add nondeterminism to the language by adding a node representing nondeterministic choice, for example:

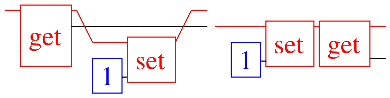


Note that nondeterministic choice nodes are drawn differently to other nodes. This is because other nodes can be duplicated or discarded (for example all of the first examples are considered equal) but nondeterminism nodes cannot (for example the two nondeterministic expressions are not equal since the former might evaluate to 2 where the latter can only evaluate to 1 or 4). We shall call nodes which can be duplicated and discarded value nodes, and nodes which cannot central nodes (the terminology will be explained below).

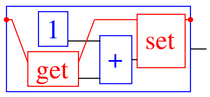
We can add imperative statements to set an integer variable and get its value, but these statements cannot be drawn in the same fashion as the others, since order of evaluation is important for imperative expressions. For example, if we were to make imperative expressions central, then the following diagrams would be graph isomorphic:



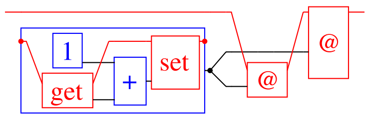
To distinguish graphs such as these, we add a new class of process nodes, and a new class of control arcs. The control arcs allow us to specify the causal order of a program, for example we can now distinguish between the graphs:



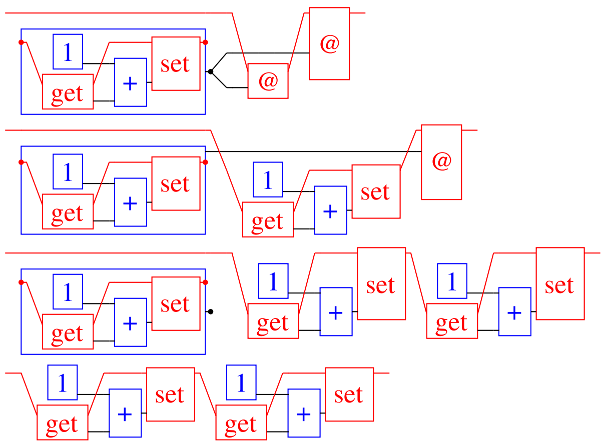
We can add higher-order functions to this graphical language by allowing function nodes which contain subgraphs, for example a function to increment the variable is:



Function application is denoted using application nodes, for example applying the increment function twice is drawn:



These graphs are viewed up to an equivalence where:

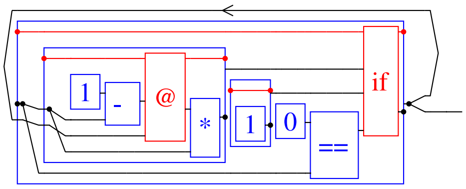


= (Beta)

= (Beta)

= (Garbage collection)

For recursive function declarations we allow cyclic graphs, for example a factorial function is:



We can give a denotational semantics for this language using domains:

- value expressions can be given a semantics in the category of complete partial orders (not necessarily with bottom) **Cpo**.
- central expressions can be given a semantics in the category of complete partial orders with binary join (to give the semantics for nondeterminism) $\mathbf{Cpo}_{\vee}$.
- process expressions can be given a semantics in the category of complete partial orders with binary join, bottom (to give the semantics of fixed points) and a state of type $\mathbf{N}$ (to give the semantics for imperative statements) $\mathrm{State}_{\mathbf{P}(\mathbf{N})}(\mathbf{Cpo}_{\perp\vee})$.

where, when $\mathbf{C}$ is a symmetric monoidal category with object X, $\mathrm{State}_X(\mathbf{C})$ is the category given by:

- Objects are objects from $\mathbf{C}$.
- Morphisms $Y \to Z$ in $\mathrm{State}_X(\mathbf{C})$ are morphisms $Y \otimes X \to Z \otimes X$ in $\mathbf{C}$.

This gives a concrete denotational semantics in particular categories of domains. Abstracting away from the details of domain theory, we discover that the structures necessary to give this denotational semantics were:

- A category V in which to interpret value expressions. Since this language has tuples and allows expressions to be duplicated or discarded, V should be a strict cartesian category (a strict symmetric monoidal category where the monoidal structure forms finite products).
- A category C in which to interpret central expressions. Central expressions have tuples which cannot be duplicated or discarded, so C should be a strict symmetric monoidal category.
- A category P in which to interpret process expressions. Process expressions have tuples which cannot be duplicated or discarded and for which evaluation order is important, so P should be a strict symmetric premonoidal category (defined by Power and Robinson for such semantics).
- We have identity-on-objects inclusions $\mathrm{V} \hookrightarrow \mathrm{C} \hookrightarrow \mathrm{P}$ which respect the product/symmetric monoidal/symmetric premonoidal structure.

- To model recursive declarations, we have a *partial trace* in V. This is an adaptation of Joyal, Street and Verity's traced monoidal categories taking account of the fact that we cannot find fixed points for every object in V. In our **Cpo** example it is only objects which have least elements which can be traced.

- To model functions we have adjunctions:

$$\mathsf{V}[X, Y \Rightarrow Z] \simeq \mathsf{V}[X \times Y, Z]$$
$$\mathsf{V}[X, Y \Rightarrow Z] \simeq \mathsf{C}[X \otimes Y, Z]$$
$$\mathsf{V}[X, Y \Rightarrow Z] \simeq \mathsf{P}[X \oslash Y, Z]$$

- To model recursive functions, we have all objects of the form $X \Rightarrow Y$ are traceable.

There are a large number of such triples, for example:

- Take V to be **Set** and C and P to be **Rel**.

- Take V to be any partially traced cartesian closed category with a commutative strong monad T, a strong monad U, both satisfying the mono requirement, and a monic natural transformation $T \to U$ which respects the monad structure. Then let C be the Kleisli category V_T and let P be the Kleisli category V_U.

- Take V, C and P to be appropriate categories of mixed data flow and control flow graphs.

Since there are so many examples of such triples of categories, it would be useful if there was an *initial* such triple. Then providing a semantics in this initial triple would be enough to give a semantics in *any* such triple.

The purpose of this paper is to show that flow graphs form the initial such triple of categories, so by giving the flow graph for a program, its semantics is given for any categorical semantics fitting the framework given above.

The paper is divided into sections:

- First, we show that an appropriate category of data flow graphs $\mathsf{Graph}(\Sigma_\mathsf{V})$ is the initial category with finite products over a signature Σ_V.

- Then we show that an appropriate category of two-coloured data flow graphs $\mathsf{Graph}(\Sigma_\mathsf{V},\Sigma_\mathsf{C})$ is the initial symmetric monoidal category over a signature Σ_C with $\mathsf{Graph}(\Sigma_\mathsf{V})$ as a sub smc.

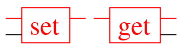
- Then we show that an appropriate category of mixed data flow and control flow graphs $\mathsf{Graph}(\Sigma_\mathsf{V},\Sigma_\mathsf{C},\Sigma_\mathsf{P})$ is the initial symmetric monoidal category over a signature Σ_P with centre $\mathsf{Graph}(\Sigma_\mathsf{V},\Sigma_\mathsf{C})$.

- Then we show that by allowing appropriate cyclic graphs we have the initial triple of categories where V is a partially traced cartesian category.
- Finally we add nodes with nested subgraphs, and show that this gives us the required closed structure.

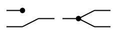
The last section is more speculative than the others, since it requires factoring the category of graphs up to beta-equivalence, eta-equivalence, and naturality. The other sections do not require such factoring, and view graphs up to an appropriate notion of bisimulation.

In each case, the main result is a soundness and completeness result, given in an appendix. These results make heavy use of the graphical presentation, which make the proofs much simpler to read. The style of the proofs should be familiar to readers with a background in process algebra, for example the normal form result for cyclic graphs is similar to Milner's proof of completeness of his axiomatization of strong bisimulation.

The observant reader may have noted that in the above example (where the premonoidal category is given as a State construction over a symmetric monoidal category) that the control line can be considered as just another data line, carrying the value of the state variable. For example we could consider the constructors:



as syntax sugar for:



This similarity is not a coincidence: the categorical basis of the graphical presentation of premonoidal categories as single-threaded graphs with one control line is that *any* premonoidal category $C \hookrightarrow P$ is a full sub-symmetric-premonoidal-category of a state transformer category $D \hookrightarrow \mathrm{State}(D)$. This result is proved in appendix.

We also provide a sketch of how two-categorical structure can be used to give an operational semantics for the graphical language, adapting Milner's semantics for action calculi.

The semantics has been implemented as a Java applet, which takes a program text and draws the corresponding flow graph (all the diagrams in this paper are drawn using this applet).

I would like to thank Adam Eppendahl, Philipa Gardner, Andy Gordon, Matthew Hennessy, Paul Levy, Rudi Lutz, Valeria de Paiva, Dusko Pavlovic, Prakash Panangaden, Eike Ritter, Edmund Robinson and Peter Selinger for discussions and suggestions.

2 Value category

2.1 Syntax

A *signature* Σ is:

- A set of sorts (ranged over by X, Y, Z).
- A set of constructors (ranged over by c, d).
- For each constructor, a source and target vector of sorts, written $c : \mathbf{X} \to \mathbf{Y}$.

Given a signature Σ_V, define the language $\mathrm{Exp}(\Sigma_V)$ as having types:

$$\begin{array}{lll} T & ::= X & \text{(Base type, } X \text{ in } \Sigma_V\text{)} \\ & \mid \quad (T,\ldots,T) & \text{(Tuple type)} \end{array}$$

expressions:

$$\begin{array}{lll} M & ::= x & \text{(Variable)} \\ & \mid \quad \mathsf{c}\ M & \text{(Value constructor, } \mathsf{c} \text{ in } \Sigma_V\text{)} \\ & \mid \quad (M,\ldots,M) & \text{(Tuple expression)} \\ & \mid \quad D\ M & \text{(Declaration binding)} \end{array}$$

declarations:

$$\begin{array}{lll} D & ::= \mathbf{let}\ P = M; & \text{(Singleton declaration)} \\ & \mid \quad D\ D & \text{(Composition of declarations)} \\ & \mid & \text{(Singleton declaration)} \end{array}$$

and patterns:

$$\begin{array}{lll} P & ::= x : T & \text{(Singleton pattern)} \\ & \mid \quad (P,\ldots,P) & \text{(Tuple pattern)} \end{array}$$

This language is similar to de Paiva and Ritter's lambda-calculus with explicit substitutions, although we have presented explicit substitutions as declarations.

In examples, we will use some syntax sugar, for example writing $1 + 2$ for $+(1(),2())$.

The novel feature of the type system for this language is to tag the judgements with a *category* $\mathbf{C}$ to determine whether the expression is a value expression, a central expression, or a process expression:

$$\Gamma \vdash M : T \text{ in } \mathbf{C}$$

For example the expression $1 + 2$ is entirely composed of value constructors, so is in the val category:

$$\vdash (1 + 2) : \text{int in val}$$

whereas $\text{set}(1 + 2)$ contains a process constructor, so is in the process category:

$$\vdash \text{set}(1 + 2) : () \text{ in proc}$$

For this section, we shall only consider value constructors, so the only category we need consider is val. Later sections will introduce the central and proc categories.

$$\mathbf{C} \ ::= \text{val}$$

With this exception, the type system is given as usual for the term algebra with declarations, using contexts:

$$\Gamma \ ::= x : T, \ldots, x : T$$

The judgements for expressions are of the form $\Gamma \vdash M : T$ in $\mathbf{C}$, and are given for variables:

$$\frac{}{\Gamma, x{:}T, \Gamma' \vdash x : T \text{ in } \mathbf{C}} \; [x \text{ not in } \Gamma']$$

value constructors:

$$\frac{\Gamma \vdash M : (B_1, \ldots, B_m) \text{ in val}}{\Gamma \vdash \mathsf{c}\; M : (C_1, \ldots, C_n) \text{ in val}} \; [\mathsf{c} : B_1, \ldots, B_m \to C_1, \ldots, C_n \text{ in } \Sigma_{\mathrm{V}}]$$

tuples:

$$\frac{\begin{array}{l}\Gamma \vdash M_1 : T_1 \text{ in } \mathbf{C} \\ \quad\vdots \\ \Gamma \vdash M_n : T_n \text{ in } \mathbf{C}\end{array}}{\Gamma \vdash (M_1, \ldots, M_n) : (T_1, \ldots, T_n) \text{ in } \mathbf{C}}$$

and declaration bindings:

$$\frac{\begin{array}{l}\Gamma \quad \vdash D : \Gamma' \text{ in } \mathbf{C} \\ \Gamma, \Gamma' \vdash M : T \text{ in } \mathbf{C}\end{array}}{\Gamma \quad \vdash D\, M : T \text{ in } \mathbf{C}}$$

Judgements for declarations are of the form $\Gamma \vdash D : \Gamma$ in $\mathbf{C}$ and are given for singleton declarations:

$$\frac{\begin{array}{l}\Gamma \vdash (P : T) : \Gamma' \\ \Gamma \ \vdash M : T \text{ in } \mathbf{C}\end{array}}{\Gamma \vdash \mathbf{let}\ P = M; : \Gamma' \text{ in } \mathbf{C}}$$

composition of declarations:

$$\frac{\begin{array}{l}\Gamma \quad \vdash D_1 : \Gamma_1 \text{ in } \mathbf{C} \\ \Gamma, \Gamma_1 \vdash D_2 : \Gamma_2 \text{ in } \mathbf{C}\end{array}}{\Gamma \quad \vdash D_1\, D_2 : \Gamma_1, \Gamma_2 \text{ in } \mathbf{C}} \quad [\Gamma_1 \text{ and } \Gamma_2 \text{ disjoint}]$$

and empty declarations:

$$\frac{}{\Gamma \vdash \ () : () \text{ in } \mathbf{C}}$$

Judgements for patterns are of the form $\Gamma \vdash (P : T) : \Gamma$ and are given for singleton patterns:

$$\frac{}{\Gamma \vdash ((x : T) : T) : (x : T)}$$

and tuple patterns:

$$\frac{\begin{array}{l}\Gamma \vdash (P_1 : T_1) : \Gamma_1 \\ \quad \vdots \\ \Gamma \ \vdash (P_n : T_n) : \Gamma_n\end{array}}{\Gamma \vdash ((P_1,\ldots,P_n) : (T_1,\ldots,T_n)) : (\Gamma_1,\ldots,\Gamma_n)} \quad [\Gamma_i \text{ have disjoint vars}]$$

2.2 Graphical semantics

The semantics of a type is given as a vector of sorts:

$$[[X]] = X \qquad [[(T_1,\ldots,T_n)]] = [[T_1]],\ldots,[[T_n]]$$

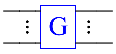
The semantics of a context is given as a vector of sorts:

$$[[x_1{:}T_1,\ldots,x_n{:}T_n]] = [[T_1]],\ldots,[[T_n]]$$

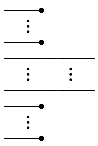
The semantics of terms is given as a flow graph (defined formally in an appendix):

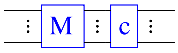
$$[[\Gamma \vdash M : T \text{ in val}]] : [[\Gamma]] \to [[T]]$$

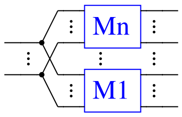
These graphs can be drawn with incoming edges on the left and outgoing edges on the right:

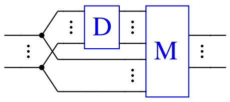


We shall usually elide the context and types where they are obvious. The semantics is defined inductively:

$[[x \text{ in val}]] =$ 

$[[\text{c } M \text{ in val}]] =$ 

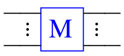
$[[(M1,\ldots,Mn) \text{ in val}]] =$ 

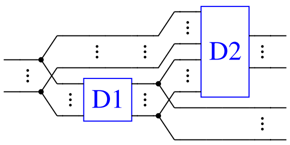
$[[D\ M \text{ in val}]] =$ 

The semantics of declarations is given as a graph:

$$[[\Gamma \vdash D : \Gamma' \text{ in val}]] : [[\Gamma]] \to [[\Gamma']]$$

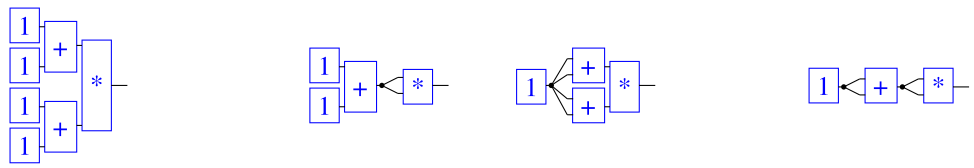
The semantics is defined inductively:

$[[\textbf{let } P = M; \text{ in val}]] =$ 

$[[D1\ D2 \text{ in val}]] =$ 

$[[\text{ in val}]] =$ 

For example, here are four graphs for the same arithmetic expression:



return ((1+1)*(1+1));	**let** x:int = 1+1; **return** (x*x);	**let** y:int = 1; **return** ((y+y)*(y+y));	**let** y:int = 1; **let** x:int = y+y; **return** (x*x);

These graphs can be proved equal using the axioms for a category with finite products, but they are *not* graph isomorphic. This means that graph isomorphism is not an appropriate equivalence on graphs if we wish to build the initial category with finite products over Σ_V. Instead, in an appendix we define an appropriate notion of bisimulation on flow graphs, and construct a category Graph(Σ_V) where:

- Objects are vectors of sorts.
- Morphisms are acyclic flow graphs, viewed up to bisimulation.

We show that Graph(Σ_V) is the initial strict cartesian category over Σ_V. Thus the graphical semantics defined above uniquely determines a semantics in *any* category with finite products over Σ_V.

3 Central category

3.1 Syntax

Given two signatures Σ_V and Σ_C with the same sorts, define the language Exp(Σ_V,Σ_C) as as extending Exp(Σ_V) with:

$$\begin{array}{lll} M & ::= & \text{...as before...} \\ & \mid & \mathrm{c}\ M \qquad\qquad (\text{Central constructor, } \mathrm{c} \text{ in } \Sigma_C) \end{array}$$

and add a new category:

$$\begin{array}{lll} \mathbf{C} & ::= & \text{...as before...} \\ & \mid & \text{central} \end{array}$$

The judgements $\Gamma \vdash M : T$ in central are as before, but we now have two new rules, one for value constructors:

$$\frac{\Gamma \vdash M : (B_1,\ldots,B_m) \text{ in central}}{\Gamma \vdash \mathsf{c}\ M : (C_1,\ldots,C_n) \text{ in central}} \quad [\mathsf{c} : B_1,\ldots,B_m \to C_1,\ldots,C_n \text{ in } \Sigma_{\mathrm{V}}]$$

and one for central constructors:

$$\frac{\Gamma \vdash M : (B_1,\ldots,B_m) \text{ in central}}{\Gamma \vdash \mathsf{c}\ M : (C_1,\ldots,C_n) \text{ in central}} \quad [\mathsf{c} : B_1,\ldots,B_m \to C_1,\ldots,C_n \text{ in } \Sigma_{\mathrm{C}}]$$

Note that the two subsumption rules are sound, one for expressions:

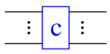
$$\frac{\Gamma \vdash M : T \text{ in val}}{\Gamma \vdash M : T \text{ in central}}$$

and one for declarations:

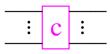
$$\frac{\Gamma \vdash D : \Gamma' \text{ in val}}{\Gamma \vdash D : \Gamma' \text{ in central}}$$

3.2 Graphical semantics

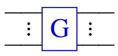
The graphical semantics of central is the same as for val, but we now have two colours of nodes. Nodes labelled with constructors from Σ_{V} are drawn:



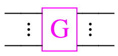
Nodes labelled with constructors from Σ_{C} are drawn:



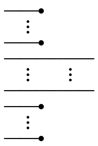
Graphs which only contain Σ_{V} nodes are drawn:

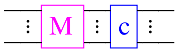


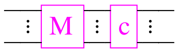
Graphs which contain Σ_{V} nodes and Σ_{C} nodes are drawn:

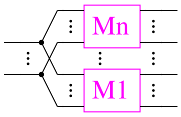


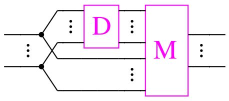
The graphical semantics of central expressions is the same as that of val expressions:

[[x in central]] = 

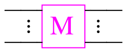
[[c M in central]] = 

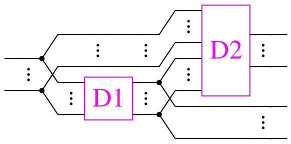
[[c M in central]] = 

[[(M_1,...,M_n) in central]] = 

[[D M in central]] = 

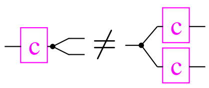
The graphical semantics of central declarations is the same as that of val declarations:

[[**let** P = M; in central]] = 

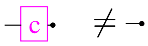
[[$D1$ $D2$ in central]] = 

[[in central]] = 

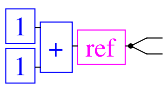
In an appendix we adapt the notion of bisimulation between graphs to require that any bisimulation is an isomorphism on central nodes. This means that for central nodes there is no natural diagonal:

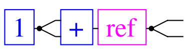


nor a natural terminal:



For example, the following graphs are bisimilar:

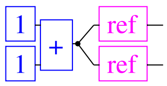




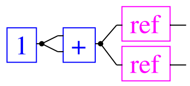
let r:ref = ref(1+1); **return** (r,r);

let y:int = 1; **let** r:ref = ref(y+y); **return** (r,r);

but they are *not* bisimilar to:



let x:int = 1+1;
return (ref(x),ref(x));



let y:int = 1;
let x:int = y+y;
return (ref(x),ref(x));

In an appendix we construct a category $\mathrm{Graph}(\Sigma_V,\Sigma_C)$ with:

$$\mathrm{Graph}(\Sigma_V) \hookrightarrow \mathrm{Graph}(\Sigma_V,\Sigma_C)$$

and show that this is the initial pair of categories:

$$\mathrm{V} \hookrightarrow \mathrm{C}$$

with:

- V a strict cartesian category over Σ_V.
- C a strict symmetric monoidal category over Σ_C.
- The inclusion an identity on objects symmetric monoidal functor.

Thus the graphical semantics defined above uniquely determines a semantics in *any* such pairs of categories.

Such pairs of categories have been studied by Benton in the form of mixed linear/non-linear logic, although he allows V and C to have different objects. The presentation here is based on Hasegawa's, although we have provided a graphical presentation rather than a term model construction. These categories form a natural model of computation where order of evaluation is unimportant, such as nondeterminism or unique name generation. In the next section we shall allow more general computations.

4 Process category

4.1 Syntax

Given three signatures Σ_V, Σ_C and Σ_P with the same sorts, define $\mathrm{Exp}(\Sigma_V,\Sigma_C,\Sigma_P)$ as extending $\mathrm{Exp}(\Sigma_V,\Sigma_C)$ with:

$$\begin{array}{lll} M & ::= \text{...as before...} & \\ & \mid \quad c\ M & \text{(Process constructor, } c \text{ in } \Sigma_P) \end{array}$$

and add a new category:

$$\mathbf{C} ::= \text{...as before...} \mid \mathsf{proc}$$

The judgements $\Gamma \vdash M : T$ in central are as before, but we now have three new rules, one for value constructors:

$$\frac{\Gamma \vdash M : (B_1,...,B_m) \text{ in proc}}{\Gamma \vdash \mathsf{c}\ M : (C_1,...,C_n) \text{ in proc}}\ [\mathsf{c} : B_1,...,B_m \to C_1,...,C_n \text{ in } \Sigma_{\mathrm{V}}]$$

one for central constructors:

$$\frac{\Gamma \vdash M : (B_1,...,B_m) \text{ in proc}}{\Gamma \vdash \mathsf{c}\ M : (C_1,...,C_n) \text{ in proc}}\ [\mathsf{c} : B_1,...,B_m \to C_1,...,C_n \text{ in } \Sigma_{\mathrm{C}}]$$

and one for process constructors:

$$\frac{\Gamma \vdash M : (B_1,...,B_m) \text{ in proc}}{\Gamma \vdash \mathsf{c}\ M : (C_1,...,C_n) \text{ in proc}}\ [\mathsf{c} : B_1,...,B_m \to C_1,...,C_n \text{ in } \Sigma_{\mathrm{P}}]$$

Again, we have the soundness of the two subsumption rules, one for expressions:

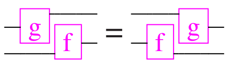
$$\frac{\Gamma \vdash M : T \text{ in central}}{\Gamma \vdash M : T \text{ in proc}}$$

and one for declarations:

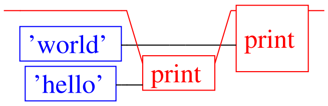
$$\frac{\Gamma \vdash D : \Gamma' \text{ in central}}{\Gamma \vdash D : \Gamma' \text{ in proc}}$$

4.2 Graphical semantics

The graphical presentation of proc is rather different from that of central and val. If we were to use the 'obvious' semantics, then we would discover that not all graph-isomorphic terms are equal. In particular, in central we have:



This is not true in proc where order of evaluation is important. We would like to retain the notion that if two terms have isomorphic graphs then they are provably equal, so we shall not use this graphical presentation. Instead we shall introduce new *control* arcs in addition to the existing data arcs. Each proc node has incoming and outgoing control arcs in addition to its data arcs, for example:



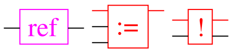
let (h:string, w:string) = ('hello', 'world');
print (h);
print (w);

The values 'hello' and 'world' come from val, so order of evaluation is unimportant. The print nodes, however, come from proc, where order of evaluation is significant, so print has an incoming and outgoing control arc as well as its incoming data arc:

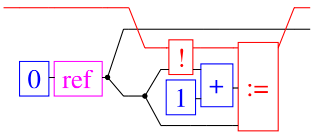
As another example, the ML reference functions can be typed (using only integer references for this example):

ref : central (int) : ref
:= : proc (ref,int) : ()
! : proc (ref) : int

or graphically:

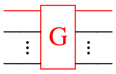


For example:

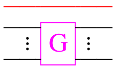


let r:ref = ref(0);
r := (1 + !(r));
return r;

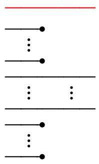
The semantics of proc terms is given as a graph with one incoming and one outgoing control edge:

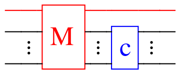


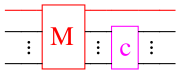
Note that we can embed central graphs into proc graphs by adding a control edge:

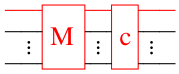


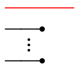
The graphical semantics of proc expressions is (we give the semantics for nullary tuples and pairs below, other tuples are similar):

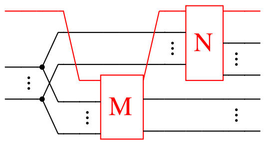
[[*x* in proc]] = 

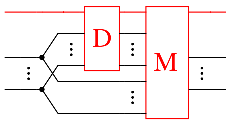
[[c *M* in proc]] = 

[[c *M* in proc]] = 

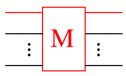
[[c *M* in proc]] = 

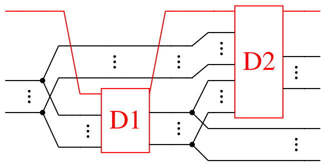
[[() in proc]] = 

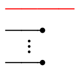
[[(*M*,*N*) in proc]] = 

[[*D M* in proc]] = 

The graphical semantics of proc expressions is:

[[**let** *P* = *M*; in proc]] = 

[[*D1 D2* in proc]] = 

[[in proc]] = 

Graphs with one incoming control arc and one outgoing control arc form a *strict symmetric premonoidal* category, a notion introduced (in slightly different form) by Power and Robinson. In an appendix we construct a category $\text{Graph}(\Sigma_V,\Sigma_C,\Sigma_P)$ of graphs with one control line, with:

$$\text{Graph}(\Sigma_V) \hookrightarrow \text{Graph}(\Sigma_V,\Sigma_C) \hookrightarrow \text{Graph}(\Sigma_V,\Sigma_C,\Sigma_P)$$

and show that this is the initial triple of categories:

$$V \hookrightarrow C \hookrightarrow P$$

with:

- V a strict cartesian category over Σ_V.
- C a strict symmetric monoidal category over Σ_C.
- P a strict symmetric premonoidal category over Σ_P.
- The inclusions are identity on objects symmetric premonoidal functors.

Thus the graphical semantics defined above uniquely determines a semantics in *any* such triples of categories.

Such triples of categories have been studied by Power and Thielecke, and compared with indexed categories. The presentation here is slightly different, in that we have presented V and C *a priori* rather than synthesizing it from P. Since central constructors play an important role in languages such as the ν-calculus, it seems natural to include C in the categorical presentation. See Selinger's discussion in his presentation of control categories. Our presentation of premonoidal categories is based on Power's presentation using **Subset**-enriched categories.

5 Partial trace

5.1 Syntax

Given three signatures Σ_V, Σ_C and Σ_P with the same sorts, where a subset of the sorts are tagged as **traceable**, define $\text{RecExp}(\Sigma_V,\Sigma_C,\Sigma_P)$ as extending $\text{Exp}(\Sigma_V,\Sigma_C,\Sigma_P)$ with:

$$\begin{array}{lll} D & ::= & \text{...as before...} \\ & | & \textbf{local rec } x;\, D \text{ (Local recursive declaration)} \end{array}$$

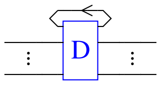
Add new judgements $\Gamma \vdash T$ **traceable**, which for this section just inherits the tags from the signatures. Then the type rule for local recursive declarations is:

$$\frac{\Gamma \vdash T\ \textbf{traceable} \qquad \Gamma, x : T \vdash D : (\Gamma_1, x : T, \Gamma_2)\ \text{in val}}{\Gamma \vdash (\textbf{local rec}\ x;\ D) : (\Gamma_1, \Gamma_2)\ \text{in val}}$$

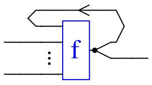
Note that recursive declarations are only allowed in val: this restriction is based on the motivating denotational model, where non-trivial fixed points only exist in **Cpo**, not in $\mathbf{Cpo}_\perp$. The restriction to **traceable** types is also based on this example: **Cpo** does not have fixed points for all objects, only those with least elements. So in this motivating example, V is **Cpo**, P is $\mathbf{Cpo}_\perp$, and the traceable objects are those with least elements.

5.2 Graphical semantics

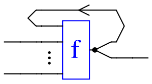
Previously, all of the graphs we have described have been acyclic. In order to give the semantics for recursive declarations, we allow cyclic graphs:

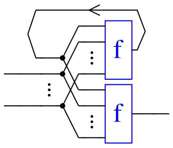
[[**local rec** x; D in val]] = 

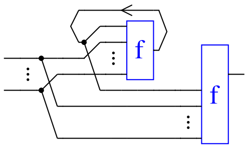
For example, the fixed point of f is (when f has a **traceable** result):

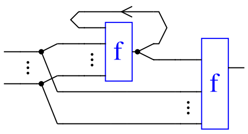


This is an indexed fixed point becase:



=  (Naturality of copy)

=  (Naturality of Tr)

=  (Indexed dinaturality of Tr)

However, not all cyclic graphs can be expressed as a program. In particular:

- We have restricted **local rec** declarations to **traceable** types, so any cyclic path must go through at least one **traceable** edge.
- We have restricted **local rec** declarations to val declarations, so any nodes in a cycle are val nodes.

Traces were initially proposed as a categorical model of cycles in knots, and this graphical presentation is just a simplified version of the knot diagrams presented by Joyal, Street and Verity. Where they were concerned with knots, we are just concerned with graphs, so we have replaced their braided monoidal setting with a simpler symmetric monoidal one.

However, our graphs do not have a trace, because of the restriction that only **traceable** sorts can be declared recursively. Instead, we use a weaker notion of *partial* traceability, where we impose the restriction that feedback edges are traceable. In the case where all types are traceable, the two notions coincide.

In an an appendix we define partially traced cartesian categories, and show that when the categories of cyclic graphs:

$$\mathrm{CGraph}(\Sigma_{\mathrm{V}}) \hookrightarrow \mathrm{CGraph}(\Sigma_{\mathrm{V}},\Sigma_{\mathrm{C}}) \hookrightarrow \mathrm{CGraph}(\Sigma_{\mathrm{V}},\Sigma_{\mathrm{C}},\Sigma_{\mathrm{P}})$$

form the initial triple of categories:

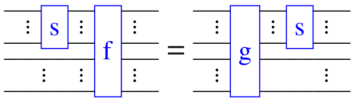
$$\mathrm{V} \hookrightarrow \mathrm{C} \hookrightarrow \mathrm{P}$$

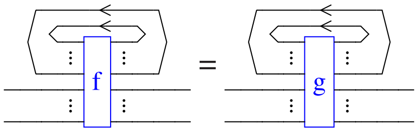
with:

- V a partially traced cartesian category over Σ_{V}.
- C a strict symmetric monoidal category over Σ_{C}.
- P a strict symmetric premonoidal category over Σ_{P}.
- The inclusions are identity on objects symmetric premonoidal functors.

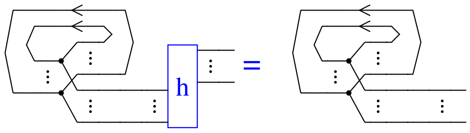
The axiomatization used for partially traced cartesian categories is more powerful that that of Joyal, Street and Verity's axiomatization for traced monoidal categories. Their axiomatization is sound and complete for graphs up to graph isomorphism, but in order to get completeness for graphs up to bisimulation, an additional property is needed (thanks to Peter Selinger for demonstrating a mistake in an earlier formulation, and for pointing out the connection with Plotkin uniformity).

A *shuffle* s is a morphism built only from composition, tensor, identity, symmetry, diagonal and terminal. A trace is *uniform wrt shuffles* whenever:

If: 

then: 

It is routine to show that Plotkin uniformity is equivalent to a trace which is *uniform wrt strict morphisms*, where a strict morphism is one such that:



It is easy to show that any shuffle is strict, and so uniformity wrt shuffles is weaker than uniformity wrt strict morphisms.

Uniformity is, unfortunately, not algebraic, and it remains to be seen whether there is an algebraic axiomatization for flow graphs.

6 Functions

6.1 Syntax

Define the language RecFun(Σ_V,Σ_C,Σ_P) as extending RecExp(Σ_V,Σ_C,Σ_P) with new types:

$$\begin{array}{lll} T & ::= & \text{...as before...} \\ & | & \mathbf{C}\; T : T \end{array}$$

and new expressions:

$$\begin{array}{llll} M & ::= & \text{...as before...} & \\ & | & \mathbf{fn}\;\mathbf{C}\; P\; \{M\} & \text{(Anonymous function)} \\ & | & M\; M & \text{(Function application)} \end{array}$$

The typing for anonymous functions is:

$$\frac{\begin{array}{ll} \Gamma & \vdash (P : T) : \Gamma' \\ \Gamma, \Gamma' & \vdash M : T' \text{ in } \mathbf{C} \end{array}}{\Gamma \quad \vdash \mathbf{fn}\;\mathbf{C}\; P\; \{M\} : (\mathbf{C}\; T : T') \text{ in } \mathrm{val}}$$

and for function application is:

$$\frac{\begin{array}{l}\Gamma \vdash M : (\mathbf{C}\; T : T') \text{ in } \mathrm{val} \\ \Gamma \vdash M' : T \text{ in } \mathbf{C}\end{array}}{\Gamma \vdash M\, M' : T' \text{ in } \mathbf{C}}$$

In particular, note that anonymous functions are always values, even if the function body is central or process. This agrees with the usual definition of normal forms for operational semantics.

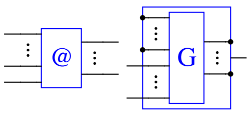
6.2 Graphical semantics

6.2.1 Cartesian closed

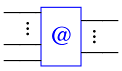
To give a graphical semantics for the cartesian closed structure, we extend the category of cyclic flow graphs $\mathrm{CGraph}(\Sigma_V)$ to that of *closed* cyclic flow graphs $\mathrm{CCGraph}(\Sigma_V)$ by allowing edges to be labelled by value function types:

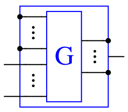
$$A,B,C ::= X \mid A_1,\ldots,A_m \Rightarrow B_1,\ldots,B_n$$

and allowing nodes of the form:

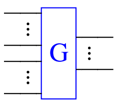


with labellings:

 $: (B_1,\ldots,B_n \Rightarrow C_1,\ldots,C_o), B_1,\ldots,B_n \to C_1,\ldots,C_o$ in $\mathrm{CCGraph}(\Sigma_V)$

 $: A_1,\ldots,A_m \to (B_1,\ldots,B_n \Rightarrow C_1,\ldots,C_o)$ in $\mathrm{CCGraph}(\Sigma_V)$

where:

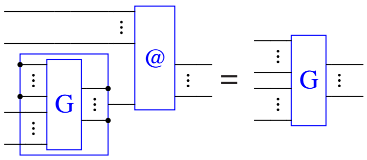
 $: A_1,\ldots,A_m, B_1,\ldots,B_n \to C_1,\ldots,C_o$ in $\mathrm{CCGraph}(\Sigma_V)$

These graphs are factored up to the equivalence required for a cartesian closed category:

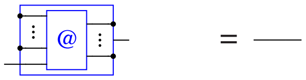
$$\mathrm{CCGraph}(\Sigma_V)[\mathbf{A}, \mathbf{B} \Rightarrow \mathbf{C}] \simeq \mathrm{CCGraph}(\Sigma_V)[\mathbf{A} \times \mathbf{B}, \mathbf{C}]$$

Graphically these axioms are:

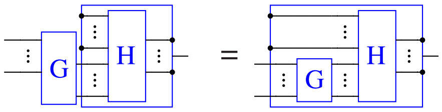
Beta:



Eta:

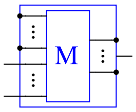


Naturality:

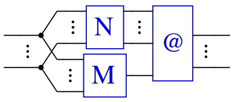


The graphical semantics is extended with:

[[**fn** val P $\{M\}$ in val]] =



[[M N in val]] =



6.2.2 Symmetric monoidal closed

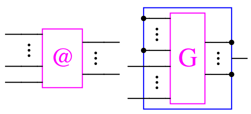
To give a semantics for the symmetric monoidal closed structure:

- we extend the category of cyclic flow graphs CGraph(Σ_V,Σ_C) to that of closed cyclic flow graphs CCGraph(Σ_V,Σ_C), and
- we extend the category CCGraph(Σ_V) to CCGraph(Σ_V,Σ_C).

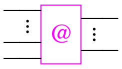
by allowing edges to be labelled by central function types:

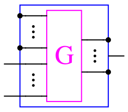
$$A,B,C ::= \ldots \mid A_1,\ldots,A_m \Rightarrow B_1,\ldots,B_n$$

We extend flow graphs to allow nodes of the form:

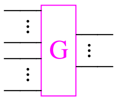


with labellings:

 : $(B_1,\ldots,B_n \Rightarrow C_1,\ldots,C_o), B_1,\ldots,B_n \to C_1,\ldots,C_o$ in CCGraph(Σ_V,Σ_C)

 : $A_1,\ldots,A_m \to (B_1,\ldots,B_n \Rightarrow C_1,\ldots,C_o)$ in CCGraph(Σ_V,Σ_C)

where:

 : $A_1,\ldots,A_m, B_1,\ldots,B_n \to C_1,\ldots,C_o$ in CCGraph(Σ_V,Σ_C)

These graphs are factored up to the equivalence required for the adjunction:

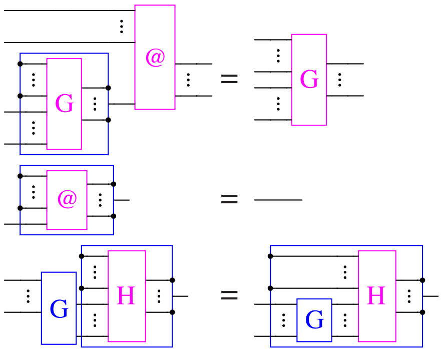
$$\text{CCGraph}(\Sigma_V,\Sigma_C)[\mathbf{A}, \mathbf{B} \Rightarrow \mathbf{C}] \simeq \text{CCGraph}(\Sigma_V,\Sigma_C)[\mathbf{A} \otimes \mathbf{B}, \mathbf{C}]$$

Graphically:

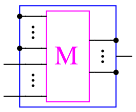
Beta:

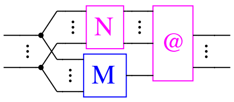
Eta:

Naturality:



The graphical semantics is extended with:

[[**fn** central P $\{M\}$ in val]] = 

[[M N in central]] = 

6.2.3 Symmetric premonoidal closed

To give a semantics for the symmetric monoidal closed structure:

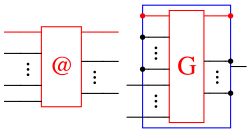
- we extend the category of cyclic flow graphs CGraph(Σ_V,Σ_C,Σ_P) to that of closed cyclic flow graphs CCGraph(Σ_V,Σ_C,Σ_P),

- we extend the category CCGraph(Σ_V,Σ_C) to CCGraph(Σ_V,Σ_C,Σ_P), and
- we extend the category CCGraph(Σ_V,Σ_C) to CCGraph(Σ_V,Σ_C,Σ_P).

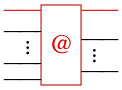
by allowing edges to be labelled with process function types:

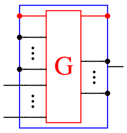
$$A,B,C ::= \ldots \mid A_1,\ldots,A_m \Rightarrow B_1,\ldots,B_n$$

and allowing nodes of the form:

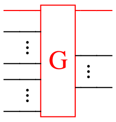


with labellings:

 : $(B_1,\ldots,B_n \Rightarrow C_1,\ldots,C_o), B_1,\ldots,B_n \to C_1,\ldots,C_o$ in CCGraph(Σ_V,Σ_C,Σ_P)

 : $A_1,\ldots,A_m \to (B_1,\ldots,B_n \Rightarrow C_1,\ldots,C_o)$ in CCGraph(Σ_V,Σ_C,Σ_P)

where:

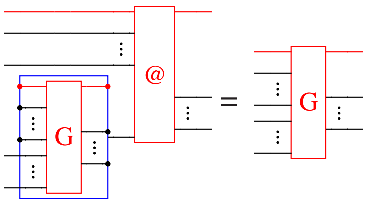
 : $A_1,\ldots,A_m, B_1,\ldots,B_n \to C_1,\ldots,C_o$ in CCGraph(Σ_V,Σ_C,Σ_P)

These graphs are factored up to the equivalence required for the adjunction:

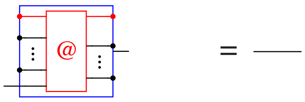
$$\text{CCGraph}(\Sigma_V,\Sigma_C,\Sigma_P)[\mathbf{A}, \mathbf{B} \Rightarrow \mathbf{C}] \simeq \text{CCGraph}(\Sigma_V,\Sigma_C,\Sigma_P)[\mathbf{A} \oslash \mathbf{B}, \mathbf{C}]$$

Graphically:

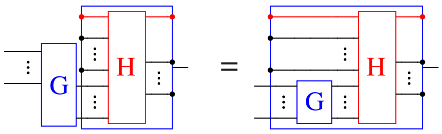
Beta:



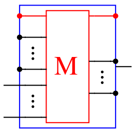
Eta:

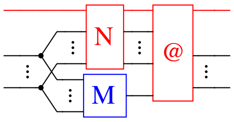


Naturality:



The graphical semantics is extended with:

[[**fn** proc P $\{M\}$ in val]] = 

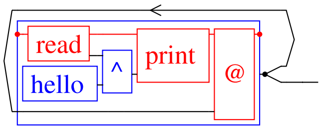
[[M N in proc]] = 

6.3 Recursive functions

To allow recursive functions, we just add a new judgement:

$$\frac{}{\Gamma \vdash \text{proc } T : T' \textbf{ traceable}}$$

For example, with appropriate string and I/O primitives, we can write a simple 'hello' program:



No change is required to the graphical semantics, except to make edges labelled with types of the form $A_1,\ldots,A_m \Rightarrow B_1,\ldots,B_n$ traceable.

6.4 Initiality

Since we have taken the initial cartesian/monoidal/premonoidal categories and factored them by equations for the closed structure, the categories:

$$\mathrm{CCGraph}(\Sigma_V,\Sigma_C,\Sigma_P) \hookrightarrow \mathrm{CCGraph}(\Sigma_V,\Sigma_C,\Sigma_P) \hookrightarrow \mathrm{CCGraph}(\Sigma_V,\Sigma_C,\Sigma_P)$$

form the initial triple of categories:

$$\mathrm{V} \hookrightarrow \mathrm{C} \hookrightarrow \mathrm{P}$$

with:

- V a partially traced cartesian category over Σ_{V}.
- C a strict symmetric monoidal category over Σ_{C}.
- P a strict symmetric premonoidal category over Σ_{P}.
- The inclusions are identity on objects symmetric premonoidal functors.
- Adjunctions:

$$\begin{array}{c} \mathrm{V}[X, Y \Rightarrow Z] \simeq \mathrm{V}[X \times Y, Z] \\ \mathrm{V}[X, Y \Rightarrow Z] \simeq \mathrm{C}[X \otimes Y, Z] \\ \mathrm{V}[X, Y \Rightarrow Z] \simeq \mathrm{P}[X \oslash Y, Z] \end{array}$$

- All objects of the form $X \Rightarrow Y$ are traceable.

Thus the graphical semantics defined above uniquely determines a semantics in *any* such triples of categories.

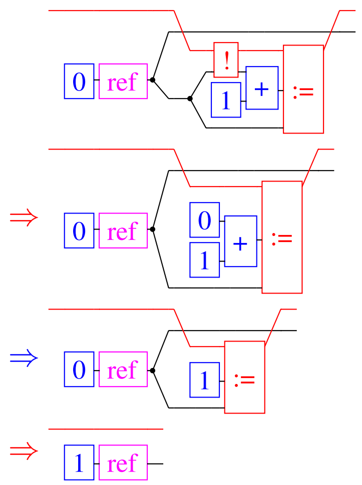
7 Operational semantics

7.1 Examples

In this paper, we have given an equational characterization of programs. Categorically, we have:

- Vectors of types are objects.
- Graphs are morphisms.

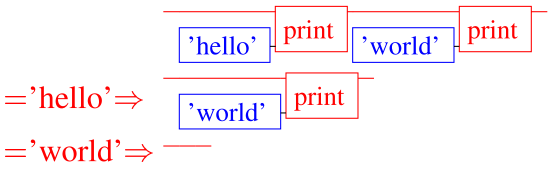
To give a treatment of the dynamics of progams, we would like to give an operational semantics, for example:



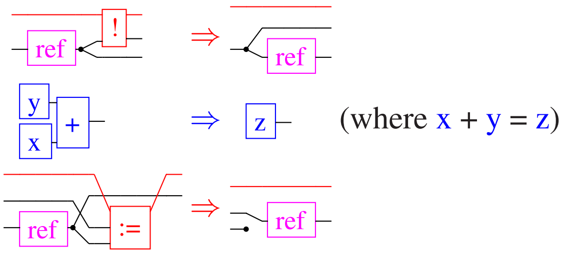
In this example, the reductions ⇒ are given as a type-preserving relation between terms. Following Milner's operational semantics for action calculi, we would expect to have the categorical picture:

- Vectors of types are objects.
- Graphs are morphisms.
- Reductions between graphs are 2-cells.

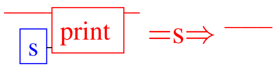
In the above case, the 2-cells are just a preorder, but in general we may be interested in labelled transition systems, which can be viewed as 2-categories where the 2-cells are labelled with an appropriate monoid, for example strings of actions. For example, we can give an lts semantics for programs containing print statements as a string-labelled 2-category:



These 2-cells are generated from atomic reductions, in these examples:

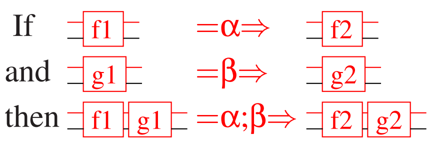


and:

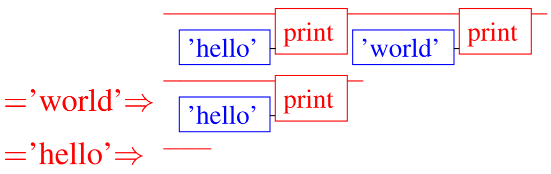


7.2 Pre-2-categories

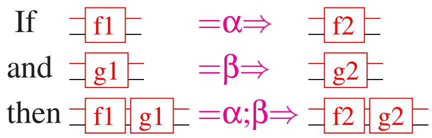
Unfortunately, we cannot just replay Milner's presentation in the premonoidal setting, since one of the axioms of a 2-category is functoriality of composition, which implies:



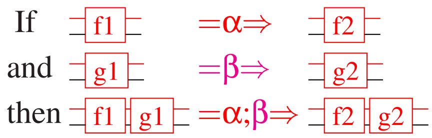
For example this would give us:



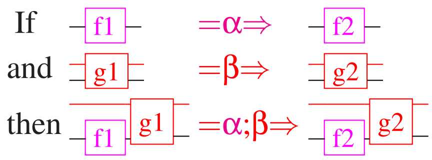
This is a similar problem to that solved by premonoidal categories, and the solution is to lift the premonoidal structure into the 2-cells. In the same way as we divided the 1-cells into *central* and *process* 1-cells, we divide the 2-cells into *central* and *process* 2-cells. Central reductions can take place anywhere in a computation, but process reductions have to happen in left-to-right order:



and:



and:



This is just the usual definition of big-step reduction to normal form, replayed in categorical language.

Formally, define $\mathsf{C} \hookrightarrow \mathsf{P}$ to be a *pre-2-category* whenever:

- C is a sub 2-category of P with the same objects.
- In addition to the usual hom-category $\mathsf{P}[X \Rightarrow Y]$ (the *central* hom-category) a category $\mathsf{P}[X \Rightarrow Y]$ (the *process* hom-category).
- $\mathsf{P}[X \Rightarrow Y]$ is a subcategory of $\mathsf{P}[X \Rightarrow Y]$ with the same objects (that is the same 1-cells).
- Two functors:

$$;_L : \mathrm{P}[X \Rightarrow Y] \times \mathrm{P}[Y \Rightarrow Z] \to \mathrm{P}[X \Rightarrow Z]$$
$$;_R : \mathrm{C}[X \Rightarrow Y] \times \mathrm{P}[Y \Rightarrow Z] \to \mathrm{P}[X \Rightarrow Z]$$

such that:

$$\begin{tikzcd}
\mathrm{P}[X \Rightarrow Y] \times \mathrm{P}[Y \Rightarrow Z] \arrow[r] \arrow[d, hook] & \mathrm{P}[X \Rightarrow Z] \arrow[d, hook] \\
\mathrm{P}[X \Rightarrow Y] \times \mathrm{P}[Y \Rightarrow Z] \arrow[r] & \mathrm{P}[X \Rightarrow Z]
\end{tikzcd}$$

and:

$$\begin{tikzcd}
\mathrm{C}[X \Rightarrow Y] \times \mathrm{P}[Y \Rightarrow Z] \arrow[r, hook] \arrow[d, hook] & \mathrm{C}[X \Rightarrow Y] \times \mathrm{P}[Y \Rightarrow Z] \arrow[d] \\
\mathrm{P}[X \Rightarrow Y] \times \mathrm{P}[Y \Rightarrow Z] \arrow[r] & \mathrm{P}[X \Rightarrow Z]
\end{tikzcd}$$

commute.

A *pre-2-functor* is a commuting square of functors respecting the pre-2-categorical structure.

Note that any 2-category is automatically a pre-2-category, since we can take C and P to be the same 2-category, and identify all of the hom-categories and composition functors.

We can then replay the definition of a premonoidal category at the 2-level to get a categorical presentation of a language with its operational semantics.

A *premonoidal pre-2-category* is a pre-2-category $\mathrm{C} \hookrightarrow \mathrm{P}$ with:

- C is a symmetric monoidal 2-category.
- Two pre-2-functors:

$$\oslash : \mathrm{C} \times \mathrm{P} \to \mathrm{P}$$
$$\obslash : \mathrm{P} \times \mathrm{C} \to \mathrm{P}$$

  such that:

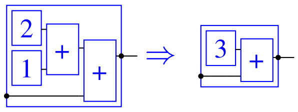
  - the three pre-2-functors $\otimes$, $\oslash$ and $\obslash$ coincide on objects,
  - the three 'obvious' pre-2-functors from $\mathrm{C} \times \mathrm{C}$ to P coincide, and
  - the symmetry in C is a natural isomorphism $X \oslash Y \simeq Y \obslash X$ in P.

Following Power's presentation of premonoidal categories as **Subset**-enriched categories, we can present pre-2-categories using enriched categories. Define a category to be *centralized* if some of its objects are tagged as central, some of its morphisms are tagged as central, and satisfying the property that if $f : X \to X'$ and X is central, then f and X' are central. Let **Central** be the category of centralized categories with functors respecting the central structure. **Central** has an asymmetric monoidal structure where $C * D$ is the subcategory of $C \times D$ where for any pair of morphisms $(f,g) : (X,Y) \to (X',Y')$ either X' is central or g is central. Then the above definition of a pre-2-category is equivalent to a **Central**-category with composition given as a functor $\mathrm{P}[X,Y] * \mathrm{P}[Y,Z] \to \mathrm{P}[X,Z]$. We can then replay Power's definition of a premonoidal category replacing **Subset**-**Cat** with **Central**-**Cat** to find a definition of a premonoidal pre-2-category equivalent to the above.

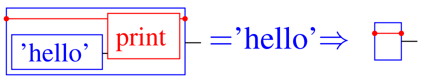
For the closed structure, the isomorphisms:

$$\begin{array}{c}
\mathrm{V}[X \Rightarrow (Y \Rightarrow Z)] \simeq \mathrm{V}[(X \times Y) \Rightarrow Z] \\
\mathrm{V}[X \Rightarrow (Y \Rightarrow Z)] \simeq \mathrm{C}[(X \otimes Y) \Rightarrow Z] \\
\mathrm{V}[X \Rightarrow (Y \Rightarrow Z)] \simeq \mathrm{P}[(X \oslash Y) \Rightarrow Z]
\end{array}$$

allow value and central reductions to take place under function bodies, but not process reductions. For example we allow:



but not:



As two extremes, we have:

- categories where the only central 2-cells are the trivial identity reductions, corresponding to languages where no reduction is allowed under lambda, and
- categories where the central and process 2-cells are the same, corresponding to langauges where reduction is allowed in any sub-term.

Thus the division of 2-cells into central and non-central categories gives a natural description of both the call-by-value lambda calculus with reduction allowed anywhere, and the canonical left-to-right reduction strategy.

8 Comparisons

8.1 Action calculi

The graphical presentation of programming most similar to that given here is Milner's *action calculi*. This is a framework for describing concurrent languages, based on symmetric monoidal 2-categories.

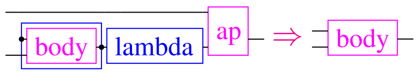
Since they are based on symmetric monoidal 2-categories, they are similar to the central categories presented here, but in action calculi the nodes (called *controls*) may contain sub-graphs.

For example, the lambda-calculus is given as having controls:

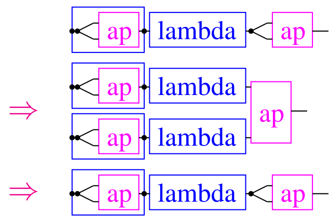
- lambda for building lambda-abstractions (originally written $\lceil \_ \rceil$) This control has one sub-graph (the body of the function), no incoming arcs, and one outgoing arc (the function).
- ap for function application. This control has no sub-graphs, two incoming arcs (the function and its argument), and one outgoing arc (the result).

We can draw these (this graphical presentation is slightly different from Milner's, to make comparisons with this paper simpler) as:

The operational semantics is given by the reduction:

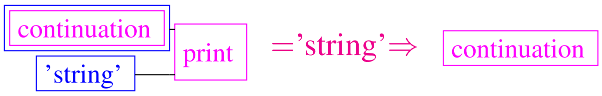


Reductions can take place anywhere in a graph, but not inside controls, for example the reduction of $(\lambda x.xx)\ (\lambda x.xx)$ is given by:

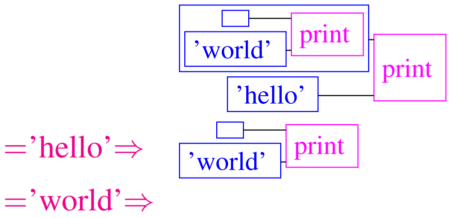


A simple language of print statements can be modelled with controls for string constants and a print control, containing a subgraph for the continuation:

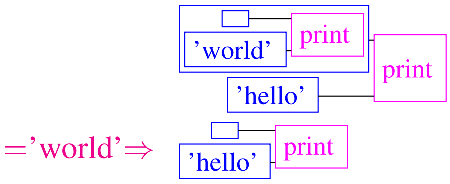
The operational semantics can be given as a 2-category where the 2-cells are strings, generated by the reduction:



For example:



Since reduction is not allowed inside controls, we do *not* have:



In action calculi, control over reduction order is achieved by the use of sub-graphs inside controls, where in our presentation reduction order is given by control arcs.

Categorically, action calculi are based on symmetric monoidal 2-categories rather than premonoidal pre-2-categories, so the categorial presentation is simpler. There is a trade-off here between the simpler categorical presentation, or the simpler graphical view given by control arcs.

To see the difference between these two presentations categorically, note that there are two ways to embed a premonoidal category into a monoidal category. The presentation here is based on an embedding into state transformers, described in an appendix. But it is also possible to use a CPS translation, in which any computation whose evaluation is delayed is placed in a continuation. Graphically, this corresponds to boxing the delayed computations, in a manner similar to the graphs above. Whether this informal connection can be formalized is left for future work.

8.2 Lambda calculus with cyclic definitions

In his thesis, Hasegawa presents a similar categorical model for lambda-calculi with explicit sharing. His setting is:

- A category with finite products V.
- A traced symmetric monoidal category C.
- An identity-on-objects inclusion $\mathrm{V} \hookrightarrow \mathrm{C}$.
- Closed structure given by an adjunction:

$$\mathrm{V}[X, Y \Rightarrow Z] \simeq \mathrm{C}[X \otimes Y, Z]$$

This is very similar to the setting described here, with two important differences:

- The addition of the premonoidal category P.
- In Hasegawa's setting, there is a trace on C rather than a partial trace on V.

The first difference is the most important: Hasegawa's graphs are pure data flow graphs, and do not require control edges. The work presented here generalizes his work from the monoidal case to the premonoidal case.

The second difference is an orthogonal issue, caused by different motivating examples. Hasegawa's motivating denotational example is the cartesian closed category **Dom**, where all objects have least elements, but maps are not necessarily strict: this provides a semantics for *call by name* or *call by need* languages. Our motivating example is the categories **Cpo** $\hookrightarrow$ $\mathbf{Cpo}_\perp$ where **Cpo** only has a partial trace: this provides a semantics for *call by value* languages.

9 Future work

9.1 Operational semantics

The operational semantics discussed in the previous section is still preliminary, and needs more work. For example, there should be a direct graphical presentation where $\mathrm{G} \Rightarrow \mathrm{G}'$ iff G has a subgraph G_1, there is an axiom $\mathrm{G}_1 \Rightarrow \mathrm{G}'_1$, and G' is G with G_1 replaced by G'_1.

Having defined a notion of labelled transition system for graphs, this opens up the usual questions of bisimulation, higher-order bisimulation, fully abstract semantics, and so on.

We could also investigate lax versions of the categorical structure, for example rather than having beta-equivalence, we could add a 2-cell for beta-reduction. This would add extra complexity to the categorical picture, but would fit better with the usual practice in defining operational semantics, and would allow us to remove all of the equivalences which are not graph bisimulations. This would improve the correspondence between the graphical and equational presentations: terms would be provably equal precisely when they are bisimilar.

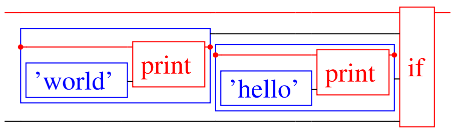
9.2 Typing issues

There are various typing issues left for future work, such as recursive types, universal and existential polymorphism, and subtyping.

Also, tracing is currently restricted to value declarations. Whilst this is adequate for the motivating example of recursive functions, there are natural examples (such as building cyclic ref-structures) where it would be appropriate to allow tracing in central declarations as well. This is left for future work.

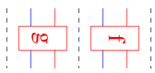
9.3 Coproducts

Currently, if-statements are only supported through thunks to delay the evalutation of the result until the value of the guard is known, for example:

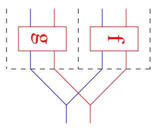


A better approach would be to add coproducts to each category, respected by each of the inclusions $V \hookrightarrow C \hookrightarrow P$. We can easily add case-statements to the language to incorporate coproducts, but finding a graphical representation is slightly trickier. The coproduct structure is monoidal, so we can represent it graphically, however we need to distinguish between the coproduct graphical structure and the premonoidal graphical structure. One possible graphical representation is (giving the version in P since the others are simpler) as follows.

The morphsim $f + g$:

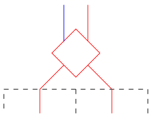


The mediating morphism $f + g$:

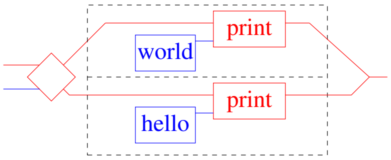


The injections inl and inr:

If we also add a graphical representation for cond : bool → 1+1 as:



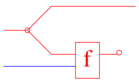
then we have a slightly better representation for closed if-statements such as:



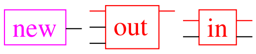
Finding a good representation for open if-statements and case-statements is left for future work.

9.4 Concurrency

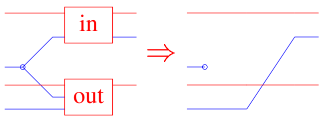
All of the computations we have looked at in this paper have been single-threaded, since there is only one control arc running through the graph. There is an obvious generalization to multi-threaded computations, very similar to action calculi, where we consider graphs with more than one control line. We could then add concurrent features to the langauge such as process forking:



With the appropriate asynchronous pi-calculus constructors:



we can then add the asynchronous pi-calculus, with operational semantics generated from:



Although this presentation is graphically appealing, it is not obvious what the categorical presentation should be. This is left for future work.