

Electronic Notes in Theoretical Computer Science 20 (1999)
URL: <http://www.elsevier.nl/locate/entcs/volume20.html> 31 pages

A Categorical and Graphical Treatment of Closure Conversion

Ralf Schweimeier

*COGS, University of Sussex
Brighton BN1 9QH, UK*

<http://www.cogs.susx.ac.uk/users/ralfs/mfps99/>

Alan Jeffrey

*COGS, University of Sussex
Brighton BN1 9QH, UK*

*CTI, DePaul University
243 South Wabash Ave
Chicago IL 60604, USA*

<http://klee.cs.depaul.edu/mfps99/>

Abstract

This paper gives a formal basis for the closure conversion phase of functional programming languages with imperative features, using a graphical semantics for the language. We present normal forms of graphs, one corresponding to procedural languages, and one corresponding to object-oriented languages. Using closure conversion, we can prove normalization results for both normal forms. Thus, we obtain sound algorithms for compiling the language into either procedural or object-oriented code. We discuss efficiency issues of the translation and suggest some improvements on the algorithm.

1 Introduction

This paper describes a categorical formalization of an important step in compiling higher-order languages: *closure conversion*.

Closure conversion is a compilation step which takes nested procedures such as:

```
proc f (x : X1) : X2 {  
  rec g;  
  proc g (y : Y1) : Y2 {  
    return G (g, x, y);  
  }  
  return F (g, x);  
}
```

and lifts the nested procedures up to top level:

```

rec g_lift;
  proc g_lift (x : X1, y : Y1) : Y2 {
    return G (mkc (g_lift, x), x, y);
  }
  proc f (x : X1) : X2 {
    return F (mkc (g_lift, x), x);
  }

```

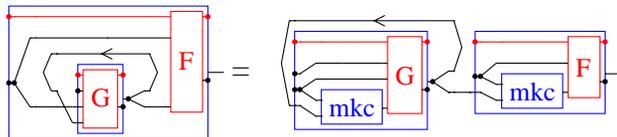
This uses a constructor $\text{mkc}(f,e)$ which builds a closure of type $X \rightarrow Y$ from a function $f : (E,X) \rightarrow Y$ and an environment $e : E$.

Closures have long been recognized as a compilation technique for functional languages such as SML, or object-oriented languages with nested class definitions such as Java 1.1. See [1] or [2] for an introduction to closures and closure conversion.

Recent work on the soundness of closure conversion has tended to be operational in nature, for example Wand and Steckler [23], Hannan [6] and Minamide, Morrisett and Harper [15].

In this paper, we propose using categorical models of computation to justify closure conversion. In particular, we propose an extension of Power and Robinson's [20] premonoidal categories, a generalization of Moggi's [17] monadic models of computation. We extend premonoidal categories with *partial closure* to model functions, based on Moggi's [16] computational lambda-calculus, and *partial traces* to model recursion, based on Joyal, Street and Verity's [12] traced monoidal categories. These extensions are similar to Hasegawa's [8] categorical semantics of letrec, Selinger's [22] co-control categories, and the models of the computational lambda-calculus by Power and Thielecke [21] and Führmann [4].

One example of this categorical model is *flow graphs*, similar to Hasegawa's [8] sharing graphs and the graphical presentation of Milner's [14] action calculi. In this graphical representation, data and control lines flow from left to right (except for recursive definitions which form loops), function bodies are represented by boxes enclosing the scope of the arguments and results, and primitives are represented by nodes. For example the above example of closure conversion is drawn:



Jeffrey [10] has shown that these graphs form the initial model, and so rather than reasoning syntactically about programs, we can reason graphically. This makes proofs much more readable (for example see the normalization proofs in the appendices) and also avoids a large number of syntactic steps which are just graph

isomorphisms.

In this paper, we:

- Sketch how partially traced, partially closed premonoidal categories can be used to model computation, and show how they can be viewed graphically.
- Formalize the notion of closure conversion as conversion into normal form.
- The first result is that with the addition of an extra axiom (a coherence condition between trace and currying) we can compile ML-like languages into C-like languages.
- The second result is that even without the extra axiom we can compile ML-like languages into Java 1.0-like languages (that is OO languages with no nested classes).
- We conclude with some discussion of efficiency issues and further work.

We provide fairly detailed proofs of these results, since we contend that the graphical presentation of programs is much more readable.

2 Graphical and categorical semantics

2.1 Premonoidal categories

Power and Robinson's [20] premonoidal categories are a generalization of Moggi's [17] monadic model of computation. Rather than present the definition of a premonoidal category directly, we shall provide a graphical presentation, which Jeffrey [10] has shown to be equivalent.

We shall present three categories of graphs, modelling three different kinds of programs:

- *Value expressions* which are guaranteed to terminate, are deterministic, and have no side effects.
- *Central expressions* which may be nondeterministic, or have side effects, as long as the order of evaluation is unimportant.
- *Process expressions* which can have any behaviour.

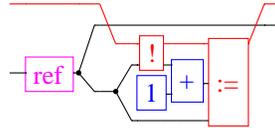
For example, in a simple programming language with assignment to integer variables:

- Constants such as `1`, `2`,... or deterministic constructors such as `+` are values.
- The constructor to create a new ref cell `ref` is a central, since it has a side-effect (allocating some memory) but order of evaluation is unimportant.
- The constructor to update `:=` or dereference `!` a ref cell are processes.

We can construct flow graphs of programs as directed acyclic graphs, where the data flow edges are labelled with types, nodes are labelled with constructors, and each edge has one source node but any number of target nodes. Since order of evaluation is important for process constructors, we add a control flow edge between them to indicate order of evaluation. For example the program:

```
let r:ref = ref(x);
    r := (1 + !(r));
r
```

has flow graph (with one incoming data line representing the free variable x , one outgoing data line representing the result r , and a control line):

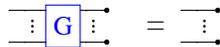


Note that we insist that each edge has a unique source, but not a unique target, to model sharing as in the example above. We view graphs up to graph isomorphism, factored by two additional equations (in [10] this equivalence is presented directly, as a form of bisimulation), where \mathbf{G} is a graph containing only value nodes.

Naturality of diagonal:



and naturality of terminal:



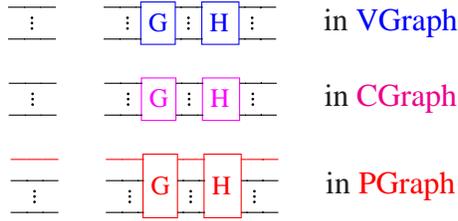
We can form categories of graphs, where:

- Vectors of sorts are objects.
- Graphs with incoming edges labelled \mathbf{X} and outgoing edges labelled \mathbf{Y} are morphisms from \mathbf{X} to \mathbf{Y} .

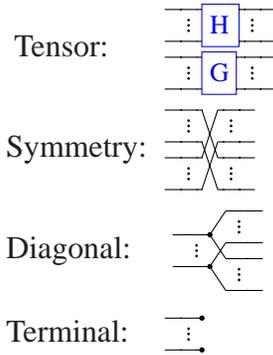
In particular, we have three categories:

- **VGraph** where the nodes are value constructors, and there are no control lines.
- **CGraph** where the nodes are value or central constructors, and there are no control lines.
- **PGraph** where the nodes are any kind of constructor, and there is one incoming and one outgoing control line.

These form categories since we have identity and composition given by:

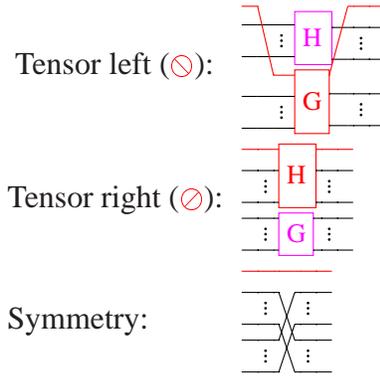


The category **VGraph** has finite products, given by:



The category **CGraph** has symmetric monoidal structure, defined similarly.

The category **PGraph** has symmetric *premonoidal* structure, since it has two natural tensor operations:



Moreover we have obvious embeddings:

$$\mathbf{VGraph} \hookrightarrow \mathbf{CGraph} \hookrightarrow \mathbf{PGraph}$$

where the inclusions respect the product/ monoidal/ premonoidal structure.

Jeffrey [10] shows that these categories of graphs are the initial such triple of categories, and so this graphical presentation is equivalent to the categorical presentation of premonoidal categories with cartesian subcategories.

2.2 *Recursive functions*

In order to be a useful model of functional programming, we need to allow recursive functions. We do this in a similar way to the graphical presentation of *control structures* by introducing function types, a new kind of value node representing function bodies, and a new kind of process node representing function application.

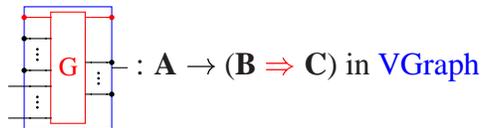
We allow edges to be labelled with function types:

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

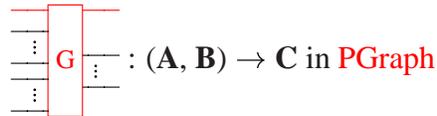
and allow application nodes:



and function nodes which capture the arguments, results and control line of the function body:



where the function body has type:

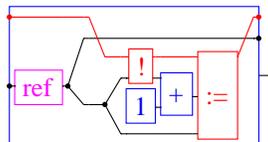


For example the function:

```

proc foo (x:int) : ref {
  let r:ref = ref(x);
  r := (1 + !(r));
  r
}
    
```

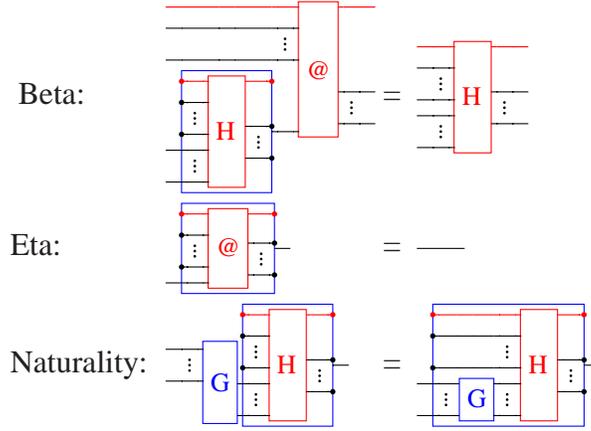
has flow graph:



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

$$\mathbf{VGraph}[A, B \Rightarrow C] \simeq \mathbf{PGraph}[A \otimes B, C]$$

Graphically (where \mathbf{G} is in \mathbf{VGraph} and \mathbf{H} is in \mathbf{PGraph}):

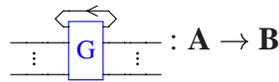


To allow for recursive functions, we allow graphs to be recursive, as long as any cycle through the graph:

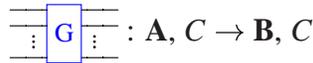
- only contains value nodes
- contains at least one edge of function type

These restrictions are equivalent to the usual syntactic restrictions on recursion in call-by-value programming languages, where recursion is only allowed over function declarations.

Such cyclic graphs (similar to Milner’s [13] reflexive action calculi) form a variant of Joyal, Street and Verity’s [12] *traced monoidal category*: their structure is recovered by making all types traceable. Graphically, a partial trace is a feedback operation:



where:



and C is a *traceable* type: in this paper we shall regard only function types as traceable.

In [10], partial traces are provided with an axiomatization, but since this axiomatization is sound and complete for graph isomorphism, we shall elide it here. In the presence of the naturality properties for diagonal and terminal, we require an additional uniformity property, but this is not required for the results which follow.

In this paper, we shall show that:

- The axioms of a partially traced, partially closed premonoidal category are enough to validate the closure conversion step of compiling a functional language to an object-oriented language without nested classes.
- The axioms of a partially traced, partially closed premonoidal category *together with an extra coherence condition* are enough to validate the closure conversion step of compiling a functional language to a procedural language.

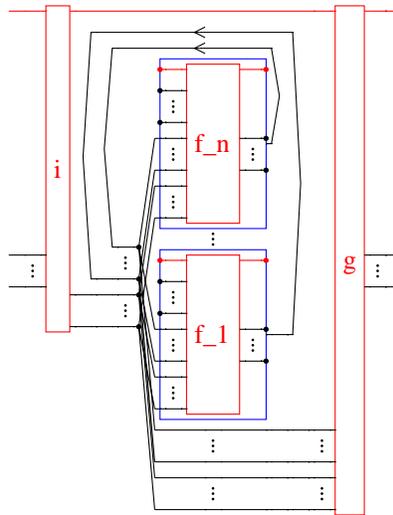
In the next section, we shall formalize this statement.

3 Level n normal forms

3.1 Definition of level n normal form

A level 0 normal form is a graph with no use of recursion or functions: we shall call such graphs *trace and lambda free (tlf)*.

A level $n+1$ normal form is a graph consisting of a number of mutually recursive functions, a tlf initialization expression i and a tlf result expression g , where the bodies of the functions, are required to be level n normal forms:

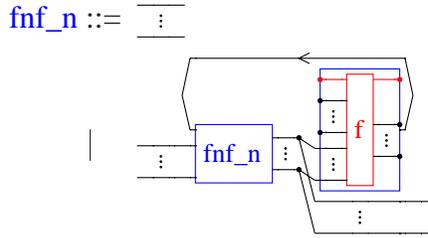


Definition (Level n normal form).

- A graph is in *level 0 normal form* if it is trace and lambda free.
- A graph from **PGraph** is in *level n normal form* if it is of the form:

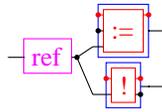
$$\text{level}_n ::= \boxed{i} \boxed{\text{fnf}_n} \boxed{g}$$

where a *level n body* is given by the grammar:



where i and g are trace and lambda free, and f is in level $(n-1)$ normal form. We can define level n normal forms for **CGraph** and **VGraph** similarly.

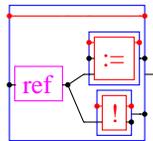
A term in level 1 normal form is a flat collection of recursive function declarations, together with some initialization and result code, so is in the form of a C-like program, or a Java-like class body. For example the level 1 graph:



corresponds (with appropriate annotations for mutable variables) to the Java class body:

```
private int r = x;
public int get () { return r; }
public void set (int x) { r = x; }
```

A function with a level 1 body is a function which expects some arguments, and returns a collection of methods which can access the parameters: this is a restricted form of a Java class definition, where the function is the constructor for the class. For example, the level 2 graph:



corresponds to the Java class definition:

```
public class Cell {
    private int r;
    public Cell (int x) { r = x; }
    public int get () { return r; }
    public void set (int x) { r = x; }
}
```

A level 2 normal form is a flat collection of recursive class declarations, together with some initialization and result code, so is in the form of a Java program.

We shall now show how an arbitrary program containing nested function declara-

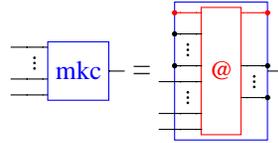
tions can be converted into level 2 normal form, using just the axioms of a partially traced, partially closed premonoidal category. We shall also show that in the presence of an additional coherence condition, any program can be converted into level 1 normal form.

3.2 Closure Conversion

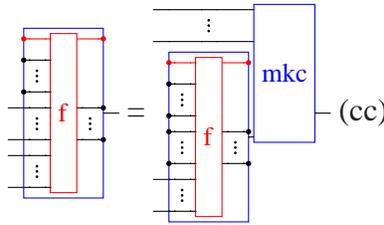
The main results of this paper show that every graph has a certain level n normal form, making use of closure conversion. We model closures by adding a new constant:

$$\text{mkc} : ((\mathbf{X}, \mathbf{Y}) \Rightarrow \mathbf{Z}), \mathbf{X} \rightarrow (\mathbf{Y} \Rightarrow \mathbf{Z})$$

defined as:



This constant satisfies the property:



The reason for adding the new constant is that we regard `mkc` as trace and lambda free, so it can be used in the bodies of level 1 normal forms.

3.3 Coherence between trace and closure

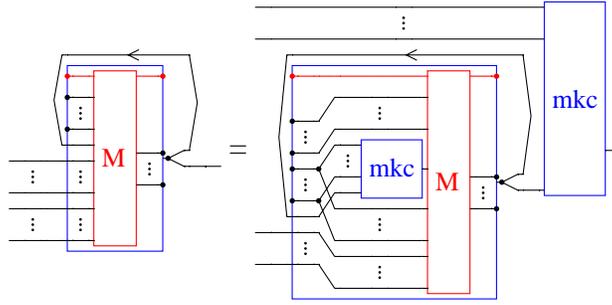
In order to prove that every graph has a level 1 equivalent, we needed to add an extra equation (tr/fn) which allows feedback loops inside functions to be lifted to top level.

Syntactically this is simple to write down:

$$\begin{aligned} & \text{fix } f . \text{fn } x . M \\ &= \text{mkc } (\text{fix } g . \text{fn } (x,y) . M [\text{mkc}(g,x) / f] , x) \end{aligned}$$

This axioms says that we can make *constant* free variables of a recursive function into *variable* parameters of a recursive function, provided we supply this function with its original actual parameters every time it is used.

Graphically this is slightly more complex:

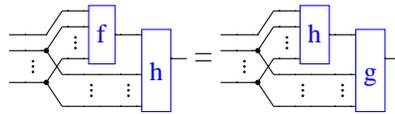


It remains an open problem whether this axiom is necessary to prove level 1 normalization (it is not necessary for level 2 normalization). In the next section, we shall show that it is sufficient.

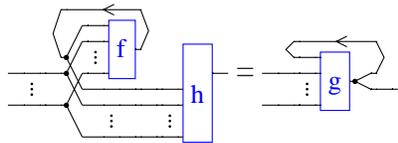
First, however, we will show that the (tr/fn) axiom is implied by the more familiar notion of *Plotkin uniformity* (see e.g. [5]).

We reformulate Plotkin uniformity in the graphical setting, in an indexed form (modelling free variables of expressions).

Definition (Plotkin uniformity). Let \mathbf{S} be a sub-cartesian category of \mathbf{VGraph} of *strict* morphisms. A partial trace on \mathbf{VGraph} is *indexed Plotkin uniform* if for any $f: X \otimes Y \rightarrow Y$, $g: X \otimes Z \rightarrow Z$ in \mathbf{VGraph} , and strict $h: X \otimes Y \rightarrow Z$ in \mathbf{S} (where Y and Z are traceable) such that



we have



Proposition. *Let mkc be strict. Then indexed Plotkin uniformity implies (tr/fn).*

Proof. See Appendix C in the full version of this paper. \square

4 Level 1 normalization

Level 1 normal forms correspond to programs where all function definitions and recursion are at top level. Thus, they can be implemented in a procedural programming language without nested functions, such as C.

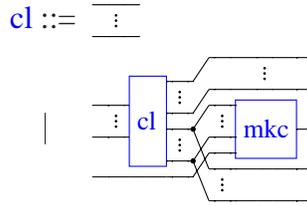
In this section, we will show that every graph is equal to a graph in level 1 normal form, using the (tr/fn) coherence condition. The proof is constructive, thus providing an algorithm for transforming any graph into a level 1 normal form.

The rest of this section will be concerned with the proof of the following theorem:

Theorem 1 (Level 1 normalization). *Every graph is provably equal to a level 1 normal form, using the (tr/fn) axiom.*

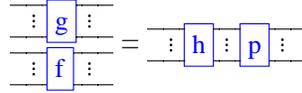
The following definition and lemmas are used in the proof of Theorem 1.

Definition. Let cl stand for a series of $mkclosures$, generated by the following grammar:



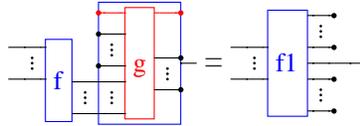
Let a *permutation* be a graph built from tensor, symmetry, identity and composition (it is easy to prove that such permutations are in 1-1 correspondence with isomorphisms on $1\dots n$).

Lemma 1 (Normalization of tensor). *Let f and g be level n bodies. Then there exists a level n body h and a permutation p such that:*



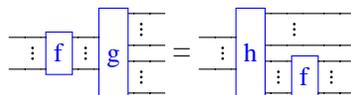
Proof. By induction on the structure of f and g . \square

Lemma 2 (Normalization of composition with a function). *Let f be a level n body, and g a level $(n-1)$ normal form. Then there is a level n body fl such that:*



Proof. Using the previous lemma, and some rewiring. \square

Lemma 3 (Normalization of composition with a tlf value). *Let f be a trace lambda free (tlf) graph in $VGraph$ and g be a level n body. Then there exists a level n body h such that:*



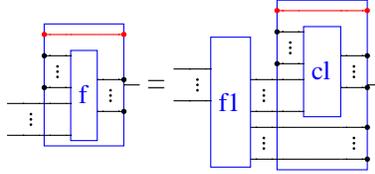
Furthermore, the numbers of functions in g and h are equal.

Proof. By copying f into the function bodies in g . \square

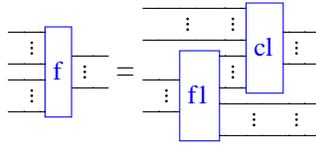
The following lemma is the heart of the proof. It shows how closure conversion can be used to lift function definitions outside an enclosing function.

Lemma 4 (Level 1 closure conversion).

(i) Let f be a level 1 body. Then there is a level 1 body $f1$ and a series of closures cl such that:



(ii) Let f be a level 1 body. Then there is a level 1 body $f1$ and a series of closures cl such that:



Proof. See Appendix A (and note that this proof uses tr/fn). \square

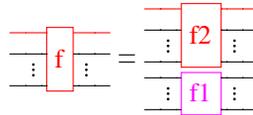
We are now able to prove Theorem 1.

Proof (of Theorem 1). Let f be a graph. The proof is by induction on the structure of f (we do not cover all cases here, but the others are similar).

(i) **Trace and lambda free:**

If f is trace and lambda free, then it is already in level 1 normal form.

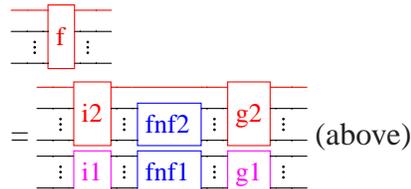
(ii) **Tensor right:**

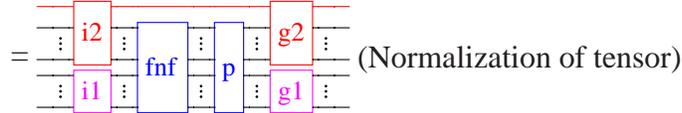


By induction, $f1$ and $f2$ have level 1 normal forms:

$$f1 = i1 ; fnf1 ; g1 \quad f2 = i2 ; fnf2 ; g2$$

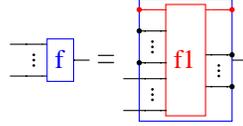
where $fnf1$ and $fnf2$ are level 1 bodies and $i1, g1, i2$ and $g2$ are tlf. Then:





where \mathbf{fnf} is a level 1 body and \mathbf{p} is a permutation. This is a level 1 normal form.

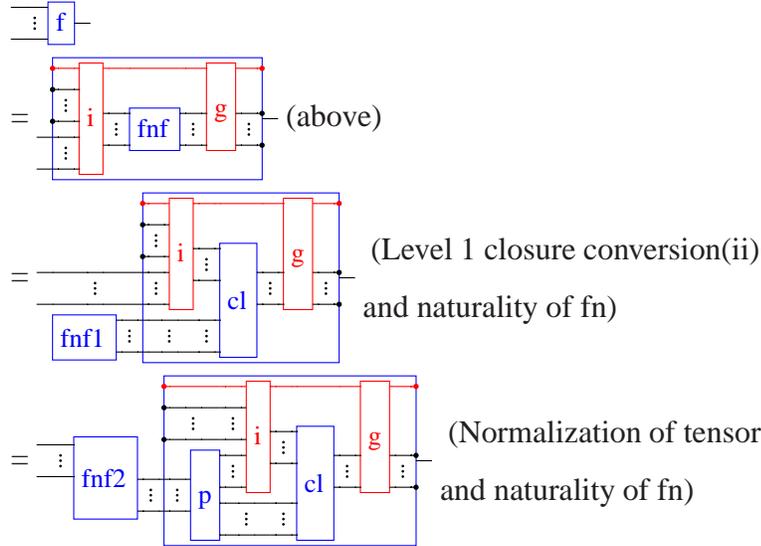
(iii) **Function:**



By induction, $\mathbf{f1}$ has a level 1 normal form:

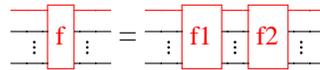
$$\mathbf{f1} = \mathbf{i} ; \mathbf{fnf} ; \mathbf{g}$$

where \mathbf{fnf} is a level 1 body and \mathbf{i} and \mathbf{g} are tlf. Then:



where $\mathbf{fnf1}$ and $\mathbf{fnf2}$ are level 1 bodies, \mathbf{p} is a permutation, and \mathbf{cl} is a series of closures. By Normalization of composition with a function, this has a level 1 normal form.

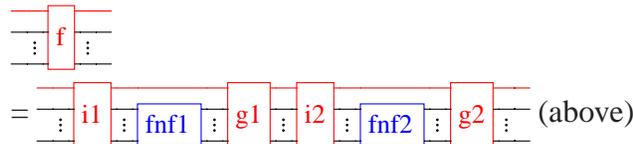
(iv) **Composition:**

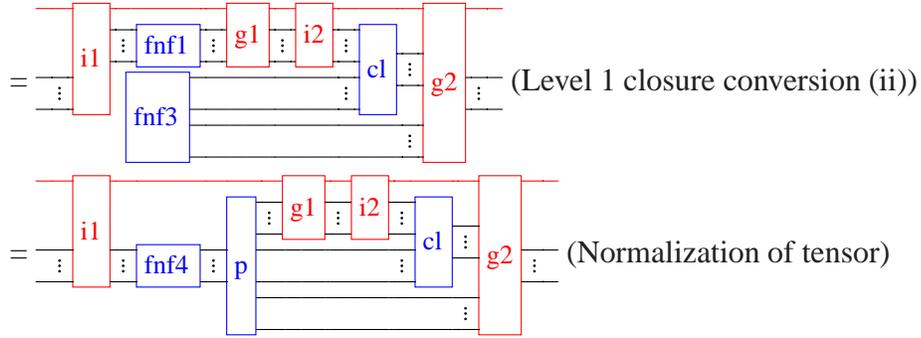


By induction, $\mathbf{f1}$ and $\mathbf{f2}$ have level 1 normal forms:

$$\mathbf{f1} = \mathbf{i1} ; \mathbf{fnf1} ; \mathbf{g1} \quad \mathbf{f2} = \mathbf{i2} ; \mathbf{fnf2} ; \mathbf{g2}$$

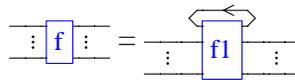
where $\mathbf{fnf1}$ and $\mathbf{fnf2}$ are level 1 bodies and $\mathbf{i1}$, $\mathbf{g1}$, $\mathbf{i2}$ and $\mathbf{g2}$ are tlf. Then:



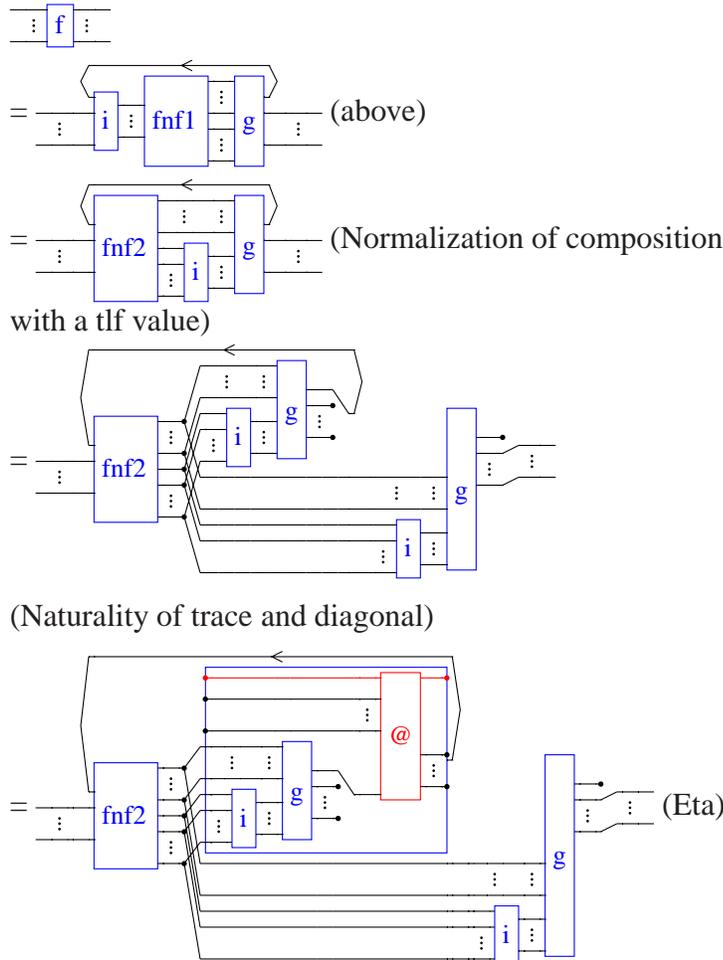


where fnf3 and fnf4 are level 1 bodies, p is a permutation, and cl is a series of closures. This is a level 1 normal form.

(v) **Trace:**



By induction $f1$ has a level 1 normal form $i;\text{fnf1};g$. Then:



where fnf2 is a level 1 body. This is a level 1 normal form.

□

5 Level 2 normalization

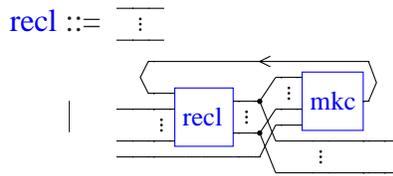
Level 2 normal forms correspond to programs with a recursive block of functions containing level 1 normal forms. Thus, they can be implemented in an object oriented programming language without nested classes, such as Java 1.0.

In this section, we will show that every graph is equal to a graph in level 2 normal form, without using the (tr/fn) coherence condition. The proof is constructive, thus providing an algorithm for transforming any graph into a level 2 normal form.

The rest of this section will be concerned with the proof of the following theorem:

Theorem 2 (Level 2 normalization). *Every graph is provably equal to a level 2 normal form, without using the (tr/fn) axiom.*

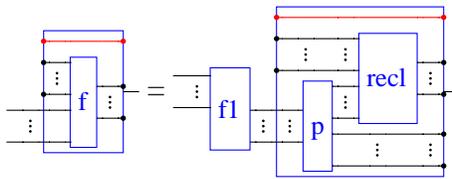
Definition. Let *recl* stand for a loop of mkclosures, generated by the following grammar:



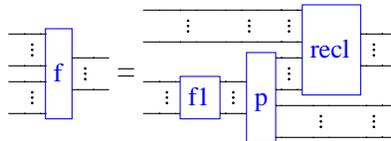
We will use the following lemmas in the proof.

Lemma 5 (Level 2 closure conversion).

(i) Let *f* be a level 2 body. Then there is an level 2 body *f1*, a loop of mkclosures *recl* and a permutation *p* such that:



(ii) Let *f* be a level 2 body. Then there is a level 2 body *f1*, a loop of mkclosures *recl* and a permutation *p* such that:



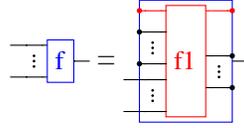
Proof. See Appendix B in the full version of this paper. Note that this proof does not use (tr/fn). \square

Lemma 6 (Normalization of loops of closures). *Every loop of closures has a level*

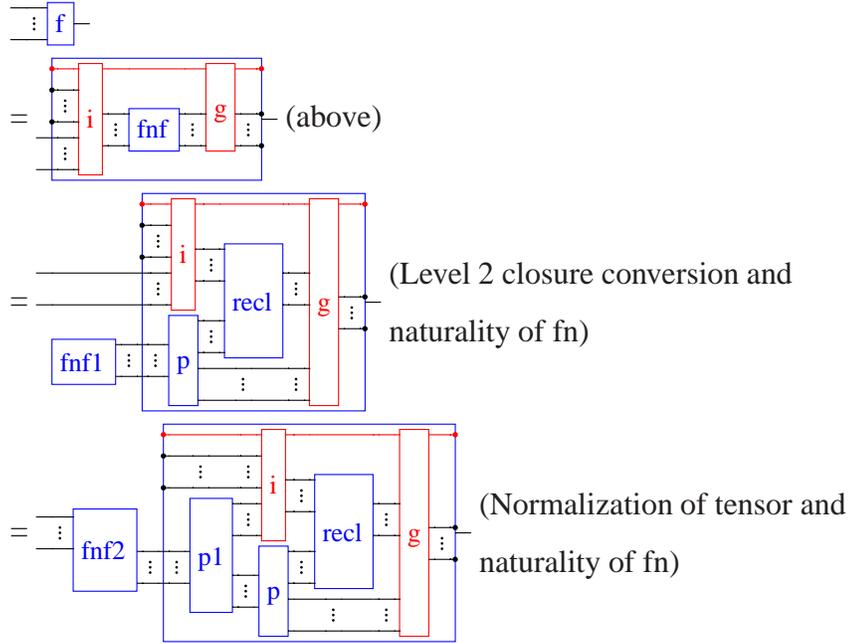
l normal form.

Proof. By eta-converting the mkclosure nodes. \square

Proof (of Theorem 2). Let f be a graph. The proof is by induction on the structure of f . All but one of the cases are similar to the level 1 case:



By induction, $f1$ has a level 2 normal form $i;fnf;g$. Then:



where $fnf1$ and $fnf2$ are level 2 bodies, p and $p1$ are permutations, and $recl$ is a loop of mkclosures.

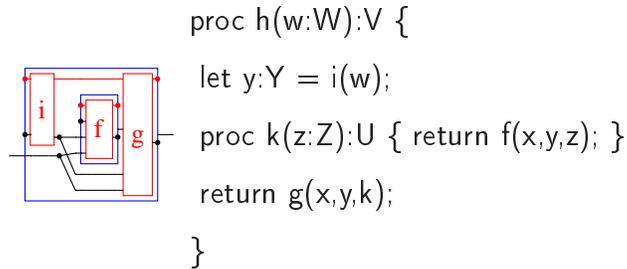
By Normalization of composition with a function and Normalization of loops of closures, this has a level 2 normal form. \square

6 Efficiency issues

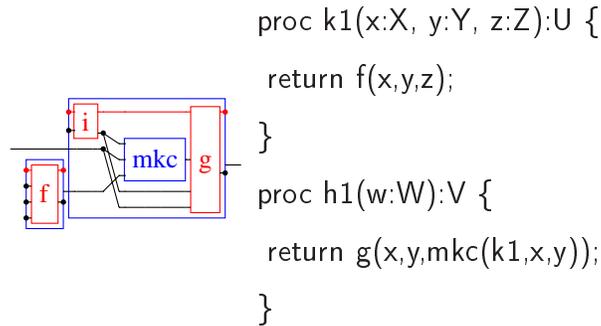
In the previous sections we have shown that the graphical semantics allows us to formalize closure conversion. We have proven that it is possible, and that it is correct. We have not, so far, talked about the quality of the translation. In this section, we describe possible future work to address some efficiency problems.

6.1 Global variables

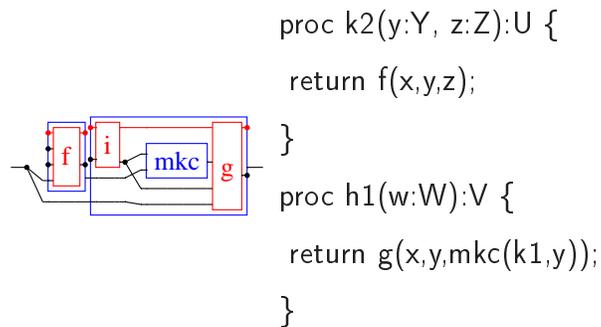
Consider a program like the following:



The function k uses a global variable x ; however, if we closure convert the program according to our algorithm, all the free variables in k will be bound, yielding the following program:



In order to lift k outside h , it would, however, be sufficient to close off only y because x occurs *free* in h . The resulting program would be



For the efficiency of a compiler, variables should not be included in a closure unless necessary. If an (entire) program contains free variables (such as library functions), we know that they will (and should) not be bound at any time. A way of ensuring this is working with *indexed graphs*.

Given a type \mathbf{X} , an \mathbf{X} -indexed graph is a graph with incoming edges of type \mathbf{X} (and possibly more). The edges in \mathbf{X} are to be understood as *global variables*, and by

working with \mathbf{X} -indexed graphs only, we ensure that

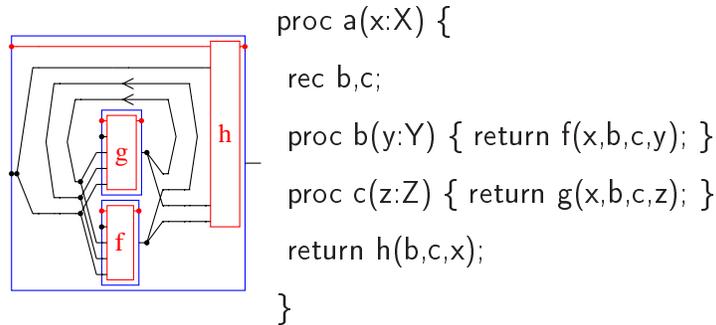
- (i) every node in the graph has access to the global variables;
- (ii) the global variables can not be bound or traced.

Categorically, indexed graphs arise as co-Kleisli categories for the comonads $\mathbf{X} \times -$, $\mathbf{X} \otimes -$ and $\mathbf{X} \circledast -$, respectively. We conjecture that these co-Kleisli categories inherit the partially traced, partially closed structure from the underlying categories, and so all of the normalization proofs still hold in this context. Thus we obtain a translation where global variables never get bound.

These co-Kleisli categories are similar to the indexed categories studied by Moggi [18] in his work on 2-level languages.

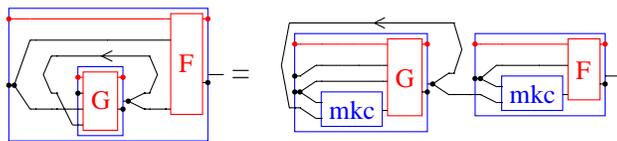
6.2 Closure Sharing

One of the reasons for investigating level 2 normal forms rather than level 1 normal forms, is that they allow for increased *closure sharing* [1, 3, 15]. For example, a function containing two mutually recursive functions is not in level 1 normal form:

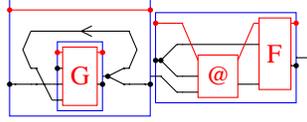


To normalize this, the two functions would have to be lifted separately out of their enclosing context, and each recursive call would require a new closure to be built. However, the above term *is* in level 2 normal form, and when executed only one closure has to be built, which is then shared between each mutual call to **f** and **g**.

For example in the introduction we showed the level 1 normalization:



This is not particularly efficient, since every recursive call to **G** causes a new closure to be built. Better would be the level 2 normalization:



It seems that level 2 normal forms represent closure sharing better than level 1 normal forms, but this is left for future work.

6.3 Closure representation

In this paper we have referred to the process of lifting functions to top level as closure conversion. In fact, this paper is part-way between closure conversion and lambda-lifting [11], since although we are explicitly representing closures as `mkc` nodes, closures have function type, and so we cannot directly extract the function and environment part from a closure.

We can still address some issues in closure representation, for example we can flatten some closures using the transformation:

$$\boxed{\text{mkc1}} \boxed{\text{mkc2}} = \boxed{\text{mkc}}$$

but this transformation can only be applied to constant closures, not to variables of closure type.

In order to perform type-safe access to the components of a closure, we would need to introduce existential polymorphism, as used by Minamide, Morrisett and Harper [15]. Then the representation of a closure of type $A \Rightarrow B$ would be $\exists a. (a, ((a, A) \Rightarrow B))$. This would separate the representation of closures from that of native functions, and it may be possible to follow Minamide, Morrisett and Harper's 'closures as objects' strategy together with Pierce and Turner's [19] use of existential types for objects to provide a better translation from functional programs to level 2 OO programs than that described here.

7 Conclusion

This paper has demonstrated that:

- Categorical models based on partially traced, partially closed premonoidal categories can be used to verify the closure conversion phase of a compiler.
- Compilation to procedural languages appears to require an extra coherence condition, allowing loops to be lifted out of functions.
- Compilation to OO languages does not require this extra condition.
- Using a sound and complete graphical model makes proofs much simpler to present.

- The categorical semantics has been implemented as a graph-drawing applet (which drew the graphs in this paper). This implementation has been extended to display the normalization process. It can be accessed online at <http://www.cogs.susx.ac.uk/users/ralfs/mfps99>.

There are a number of questions left as open problems:

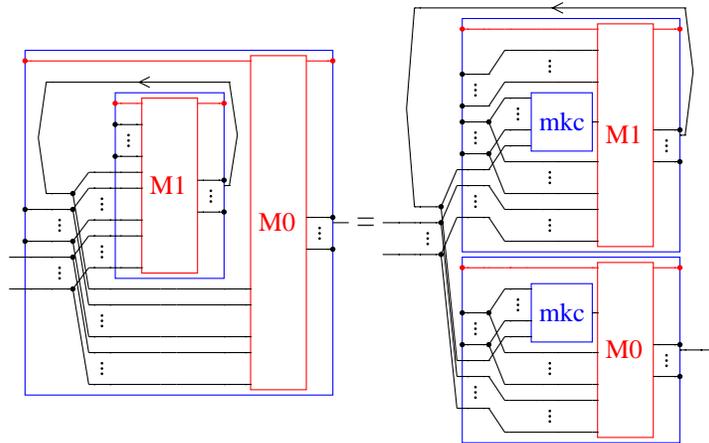
- The algorithm described by the proof given here produces very inefficient code: can we apply the same techniques to ‘industrial strength’ compilation strategies?
- Is the (tr/fn) coherence axiom necessary to the proof of level 1 normalization? If so, is there a simpler formulation with equal expressive power?
- Can the categorical semantics be extended with existential polymorphism to support the type-based analysis of explicit closures described by Minamide, Morrisett and Harper [15]?
- Hannan [6] has shown that typed closure conversion can be verified in LF [7]. Such proofs are syntactically driven, and it is not obvious how these techniques could be adapted to graphical proofs such as those described here. Does the categorical presentation of flow graphs provide a way to combine human-readable graphical proofs with machine-checkable syntax?

These problems are left as future work.

A Normalization proof - level 1 normal form

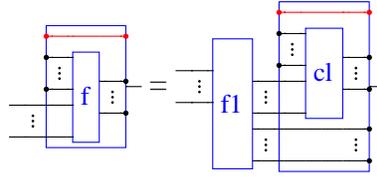
To shorten the proof, we use the following consequence of the (tr/fn) axiom, which we state without proof.

Lemma (tr/fn 2) Let $M0$ and $M1$ be graphs in \mathbf{PGraph} . Then the axiom (tr/fn) implies

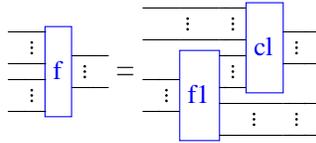


Lemma (Level 1 closure conversion)

(i) Let f be a level 1 body. Then there is a level 1 body $f1$ and a series of closures cl such that:

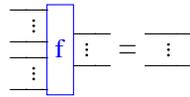


(ii) Let f be a level 1 body. Then there is a level 1 body $f1$ and a series of closures cl such that:



Proof. By induction on the number of functions in f . We prove (i) using the induction hypothesis (ii), and then prove (ii) using (i).

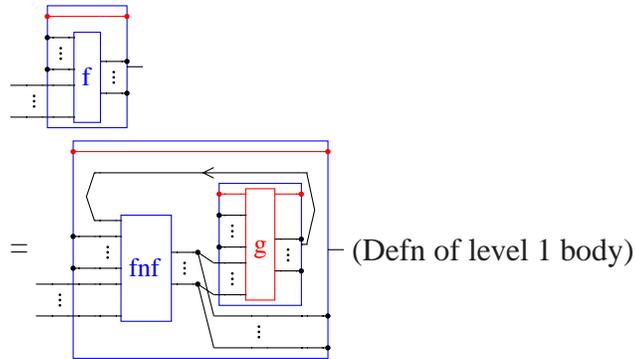
Base case:

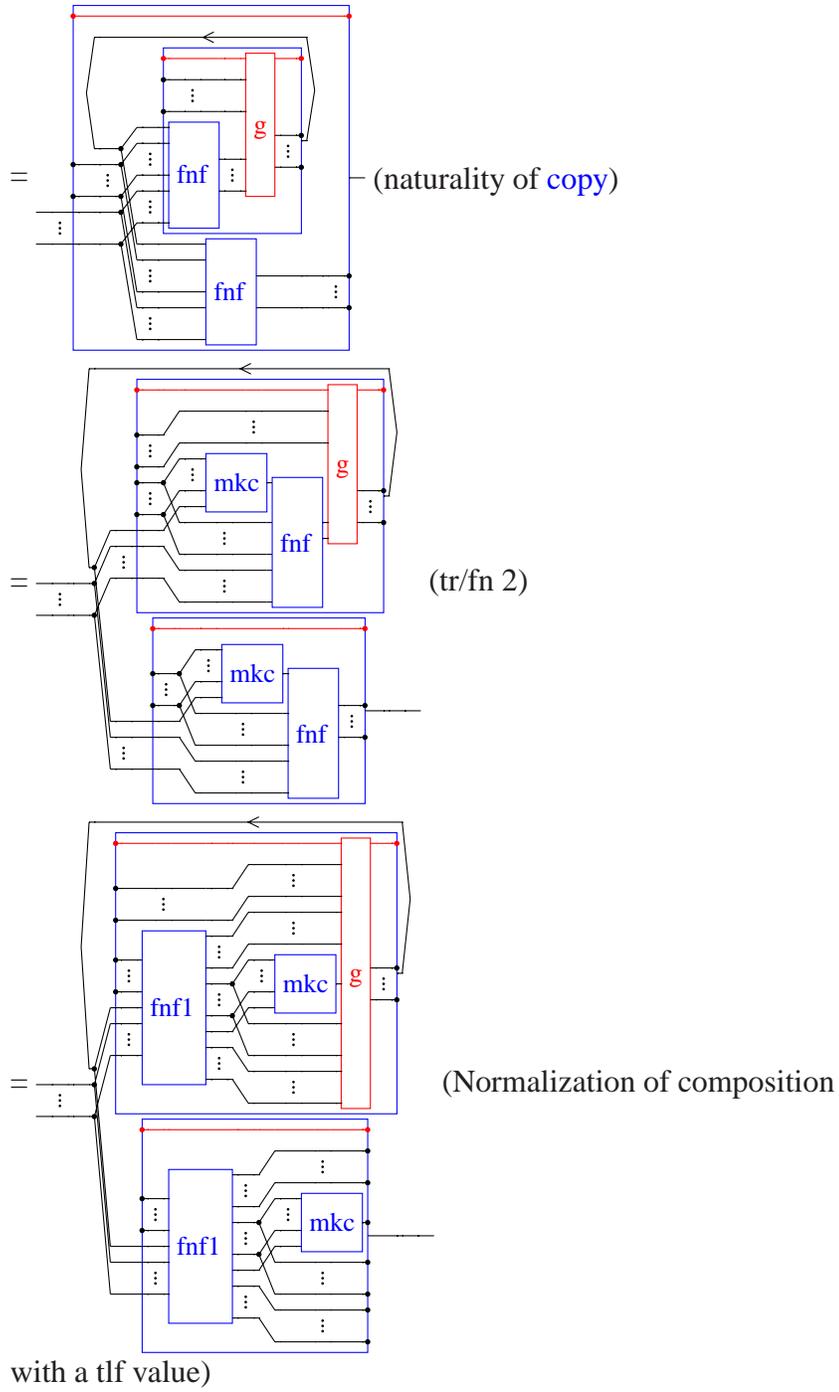


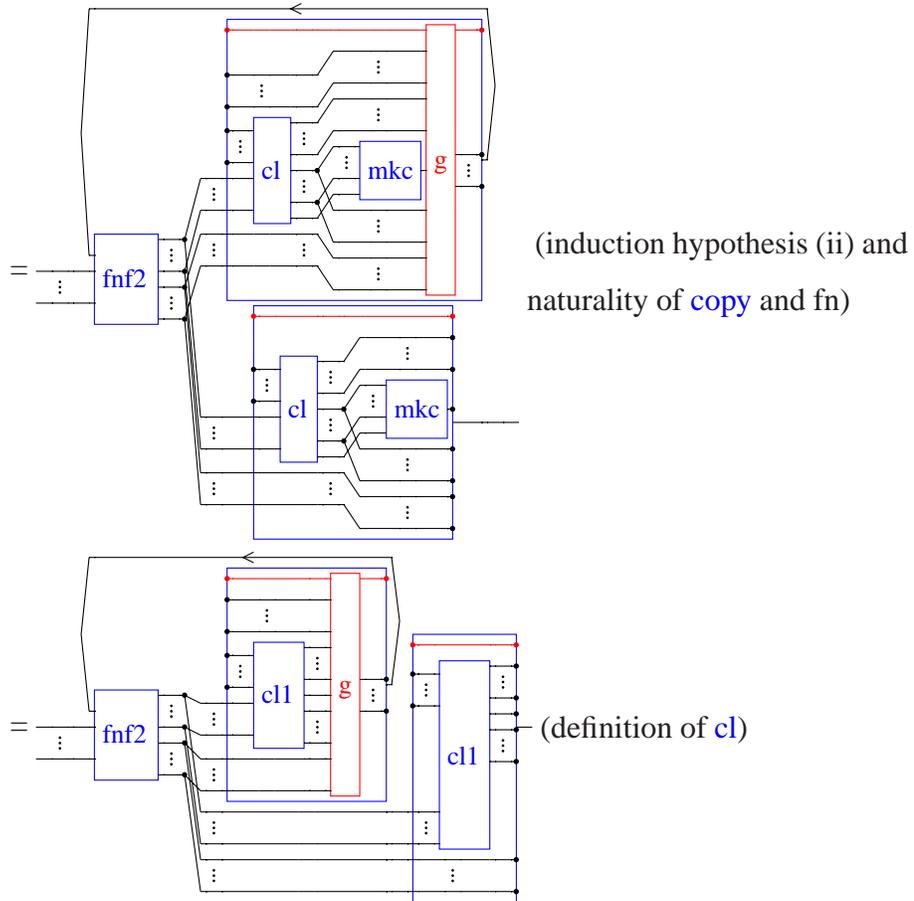
trivial, with $f1$ and cl both the identity.

Inductive step:

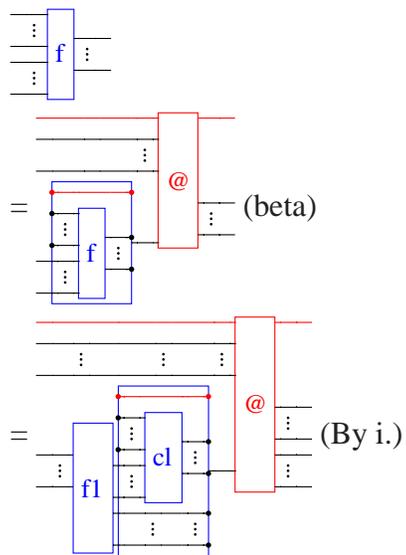
(i)

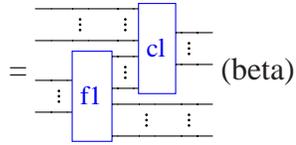






(ii)



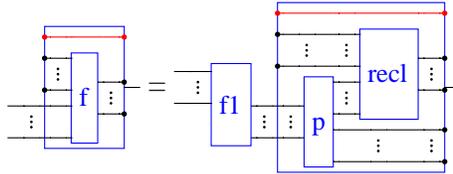


□

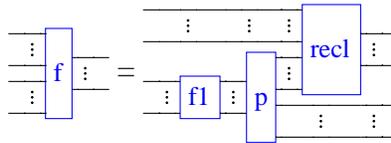
B Normalization proof - level 2 normal form

Lemma (Level 2 closure conversion).

(i) Let f be a level 2 body. Then there is a level 2 body $f1$, a loop of mkclosures $recl$ and a permutation p such that:

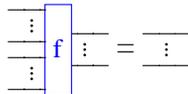


(ii) Let f be a level 2 body. Then there is a level 2 body $f1$, a loop of mkclosures $recl$ and a permutation p such that:



Proof. By induction on f . We prove (i) using the induction hypothesis (ii), and then prove (ii) using (i).

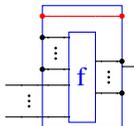
Base case:

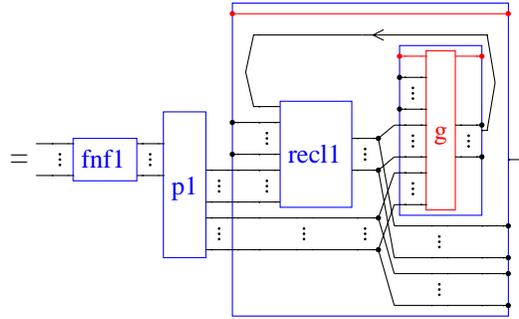
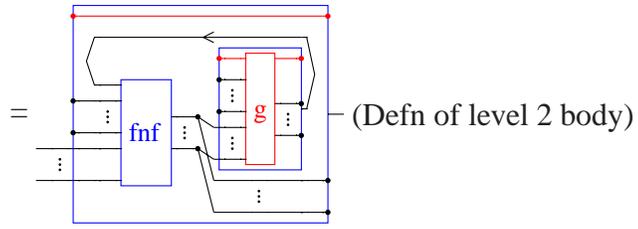


Trivial, with $f1$, p and $recl$ all the identity.

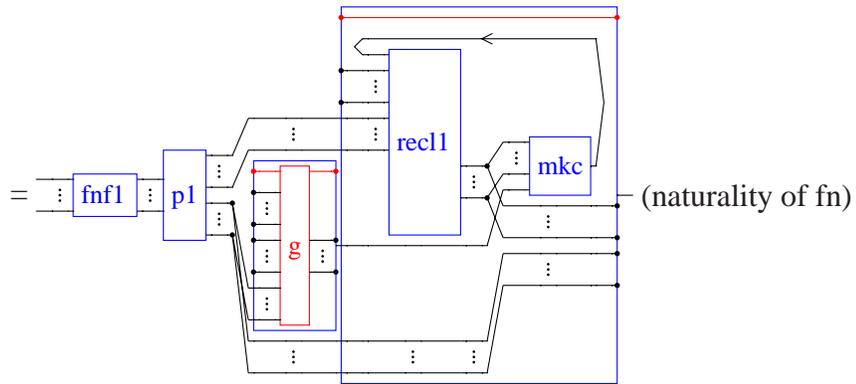
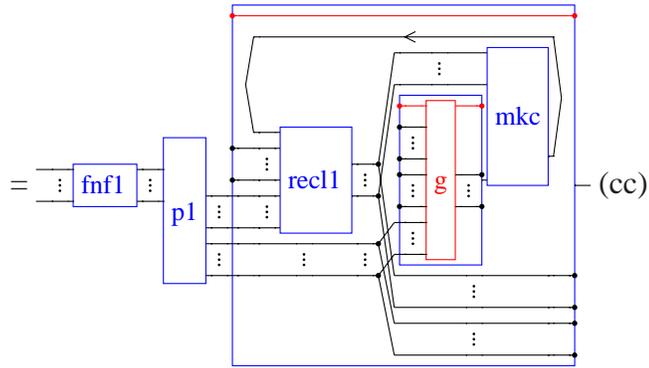
Induction step:

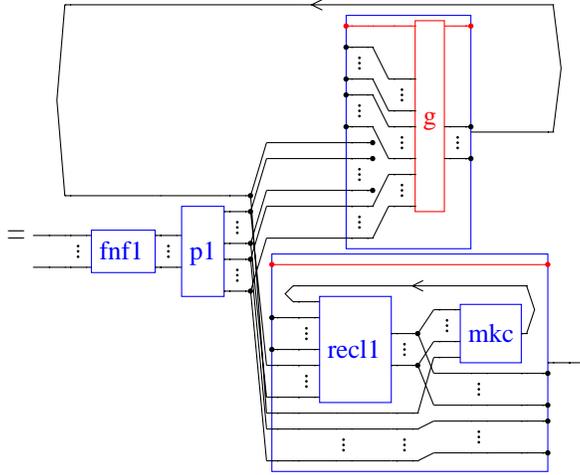
(i)



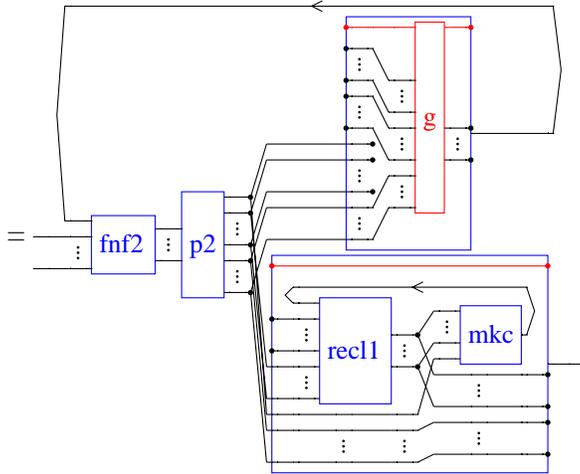


(induction hypothesis (ii) and naturality of fn)

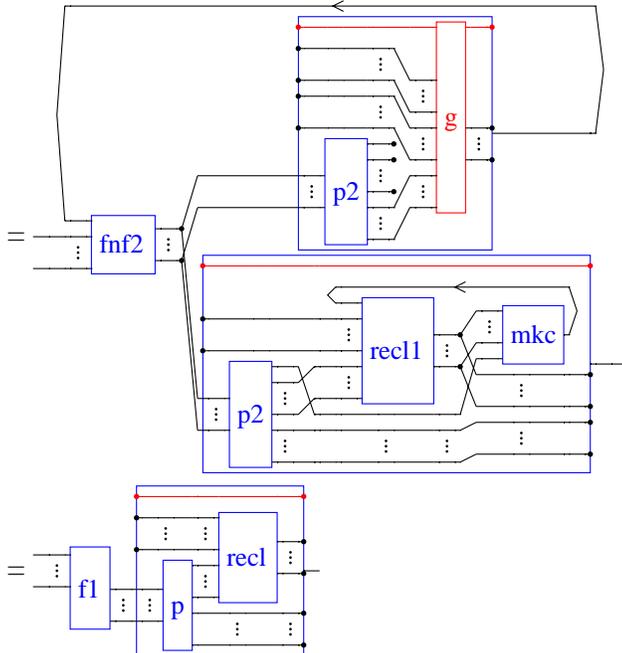




(graph isomorphism and naturality of fn)



(Normalization of tensor)



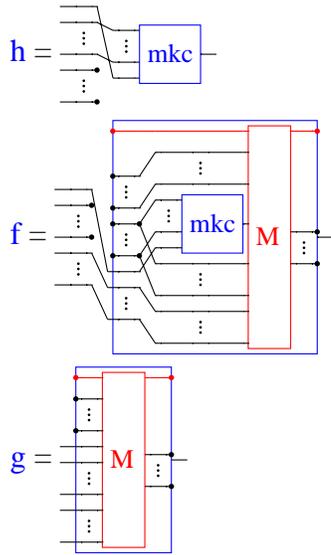
(naturality of copy and fn)

(ii) Follows as in Appendix A. \square

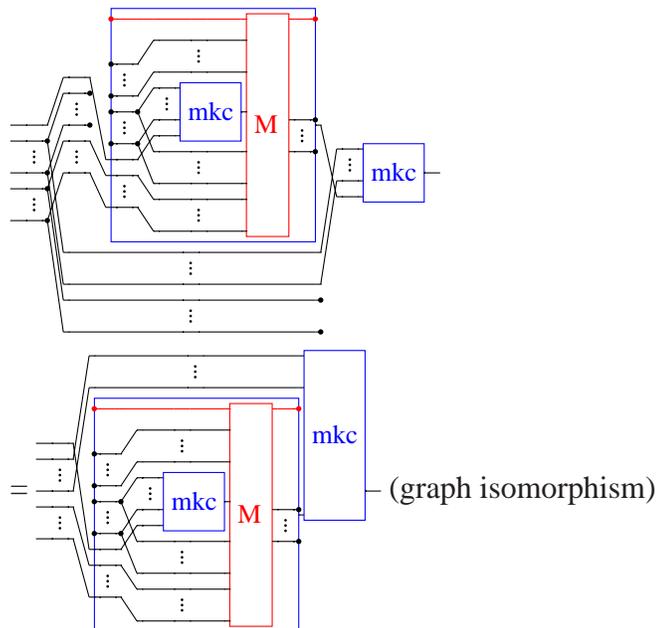
C Plotkin uniformity implies (tr/fn)

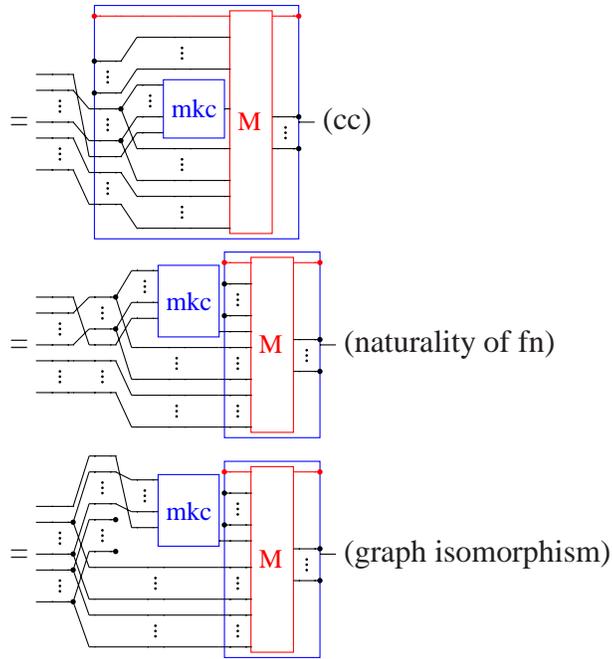
Proposition. *Let mkc be strict. Then indexed Plotkin uniformity implies (tr/fn).*

Proof. In the definition of indexed Plotkin uniformity, let



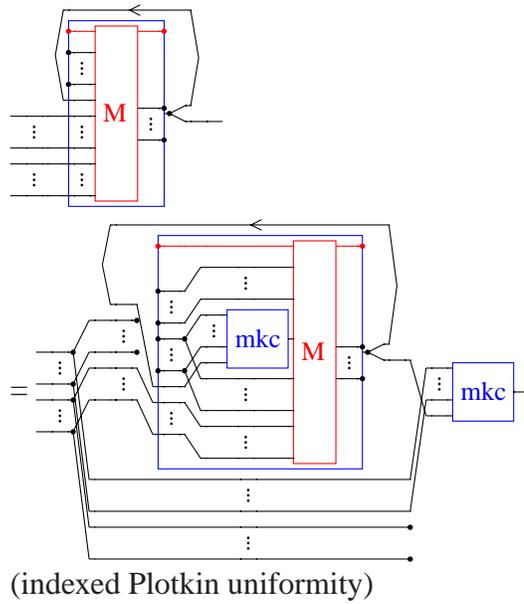
Then

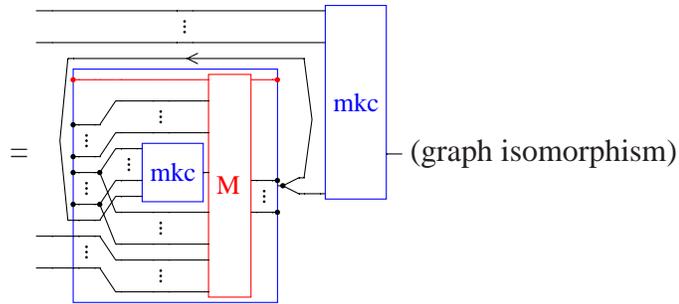




Because the strict morphisms form a cartesian category, \mathbf{h} is strict.

Using indexed Plotkin uniformity, we can then deduce (tr/fn):





□

D Bibliography

- [1] Appel, A., “Compiling with Continuations,” CUP, 1992.
- [2] Appel, A., “Modern Compiler Implementation in (Java|ML|C),” CUP, 1998.
- [3] Appel, A. and Jim, T., *Continuation-passing, closure-passing style*, in *Proc. POPL 1989*.
- [4] Führmann, C. *Direct models for the computational lambda-calculus*, in *Proc. MFPS 1999*.
- [5] Gunter, C., “Semantics of Programming Languages,” MIT Press, 1992.
- [6] Hannan, J., *Type systems for closure conversion*, in *Proc. Workshop on Types for Program Analysis*, 1995.
- [7] Hannan, J. and Pfenning, F., *Compiler verification in LF*, in *Proc. LICS 1992*.
- [8] Hasegawa, M., “Models of Sharing Graphs (A Categorical Semantics of Let and Letrec),” Ph.D thesis, Univ. Edinburgh, LFCS report ECS-LFCS-97-360, 1997.
- [9] Hasegawa, M., *Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi*, in *Proc. 3rd International Conference on Typed Lambda Calculi and Applications (TLCA’97)*, Springer LNCS 1210, pp. 196-213, 1997.
- [10] Jeffrey, A. S. A., *Premonoidal categories and a graphical view of programs*, available from <http://klee.cs.depaul.edu/premon/>, 1998.
- [11] Johnsson, T., *Lambda lifting: transforming programs to recursive equations*, in *Proc. FPCA 1985*.
- [12] Joyal, A., Street, R. H. and Verity, D., *Traced monoidal categories*, in *Math. Proc. Cambridge Philosophical Soc.* **119** (1996), pp. 425-446.
- [13] Milner, R., *Action calculi V: reflexive molecular forms*, Manuscript 1994.
- [14] Milner, R., *Calculi for interaction*, *Acta Informatica* **33** (1996), pp. 707-737.

- [15] Minamide, Y., Morrisett, G. and Harper, R., *Typed closure conversion*, in *Proc. POPL 1996*.
- [16] Moggi, E., *Computational lambda-calculus and monads*, in *Proc. LICS 1989*.
- [17] Moggi, E., *Notions of computation and monad*, in *Information and Computation* **93** (1991), pp. 55-92.
- [18] Moggi, E., *A categorical account of two-level languages*, in *Proc. MFPS 97*, *Electronic Notes in Computer Science* **6** (1997), URL <http://www.elsevier.nl/locate/entcs/volume6.html>.
- [19] Pierce, B. C. and Turner, D. N., *Simple type-theoretic foundations for object-oriented programming*, in *J. Functional Programming* **4** (1994), pp. 207-247.
- [20] Power, A. J. and Robinson, E.P., *Premonoidal categories and notions of computation*, to appear in *Math. Struct. in Comput. Science*.
- [21] Power, A. J. and Thielecke, H., *Closed Freyd- and kappa-categories*, in *Proc. ICALP 1999*.
- [22] Selinger, P., *Control categories: an axiomatic approach to the semantics of control in functional languages*, presented at MFPS 98.
- [23] Wand, M. and Steckler, P., *Selective and lightweight closure conversion*, in *Proc. POPL 1994*