From Hilbert Spaces to Dilbert Spaces: Context Semantics Made Simple

Harry G. Mairson [⋆]

Computer Science Department
Brandeis University
Waltham, Massachusetts 02454
mairson@cs.brandeis.edu

Abstract. We give a first-principles description of the *context semantics* of Gonthier, Abadi, and Lévy, a computer-science analogue of Girard's *geometry of interaction.* We explain how this denotational semantics models λ-calculus, and more generally multiplicative-exponential linear logic (MELL), by explaining the call-by-name (CBN) coding of the λ-calculus, and proving the correctness of *readback*, where the normal form of a λ-term is recovered from its semantics. This analysis yields the correctness of Lamping's optimal reduction algorithm. We relate the context semantics to linear logic types and to ideas from *game semantics*, used to prove full abstraction theorems for PCF and other λ-calculus variants.

1 Introduction

In foundational research some two decades ago, Jean-Jacques Lévy attempted to characterize formally what an *optimally efficient* reduction strategy for the λ-calculus would look like, even if the technology for its implementation was at the time lacking [13]. For the language designer, the λ-calculus is an important, canonical abstraction of the essential features required in a programming language, just as the classical physicist (or freshman physics student) views the world via a model of massless beams and frictionless pulleys, and as machine architects have the Turing machines as their essential, ideal archetype.

Lévy wanted to build a λ-calculus interpreter that was *correct* and *optimal.* Every interpreter specifies an *evaluation strategy*, determining what subexpression is evaluated next. A *correct* interpreter never chooses a subexpression whose evaluation is *divergent* unless there is no other choice, so that it produces a *normal form* (answer) if there is one. An *optimal* evaluator shares *redexes* (procedure calls) in a technically maximal sense: the problem here is that evaluation can easily *duplicate* redexes, for example in $(\lambda x.xx)((\lambda w.w)y)$.

John Lamping [12] found the algorithm that Lévy specified. Then Gonthier, Abadi, and Lévy [9, 10] made a lovely discovery: they gave a denotational semantics to Lamping's algorithm, called *context semantics*, and showed that it was equivalent to Jean-Yves Girard's *geometry of interaction* (GoI) [8]. Girard's GoI is an abstract mathematical notion, employing a lot of higher mathematics that theoretical computer scientists are not accustomed to using: Hilbert spaces,

[⋆] Supported by NSF Grants CCR-0228951 and EIA-9806718, and also by the Tyson Foundation.

C^*-algebras, and more. In contrast, context semantics is built from familiar, ordinary data structures: stacks and trees of tokens. The vectors of a Hilbert space are just a data structure, and the linear operators Girard designed were just ways of hacking the data structure. Context semantics provides a precise flow analysis, where static analysis is effected by moving *contexts* (a data structure) through a fixed program graph, so that questions like "can call site α ever call function f?" become straightforward to answer. Girard's matrix *execution formula* modulo a *path algebra* is then just transitive closure of a directed graph (just like in an undergraduate automata theory class), where the algebra rules out certain paths. The GoI formalism is not unlike the use of generating functions to analyze combinatorial problems. Computer scientists should find context semantics to be a more compelling formalism, because there is no math, and the resemblance to static program analysis is so strong.

The main purpose of this paper is to give a simple explanation of context semantics—it's the paper I wish I could have *read* six or eight years ago. I particularly want to give a first-principles proof of the algorithmic correctness of optimal reduction, via a naive explanation of *readback*: the process whereby the normal form of a λ-term is recovered from its context semantics. *A key goal is to explain how GoI, and context semantics, and games, and full abstraction, are founded on technical, detailed ways of talking about Böhm trees (term syntax);* like Molière's Monsieur Jourdain, we should know prose when we are speaking it. This observation is not meant to be dismissive—I want to explain how context semantics details the mechanics of reduction and the extraction of answers, via static analysis. I want to explain how linear types give a refined view of contexts. Most important, I want the discussion to be as informal and intuitive as possible—like a conversation—with incomplete proofs and basic intuitions.

For those who from semantics recoil, recall: this is algorithm analysis, establishing the correctness of an incremental interpreter. Related complexity analysis, both of Lamping's algorithm and the problem, can be found in [6, 4].

2 Linear $\boldsymbol{\lambda}$-terms

The best place to start—representing the essence of the full problem, but in miniature—is with *linear* λ-terms: every variable must occur exactly once. Church numerals $\lambda s.\lambda z.s^n z$ are ruled out (s may occur more than once or not at all), also Boolean truth values (recall $\mathrm{T} \equiv \lambda t.\lambda f.t$, $\mathrm{F} \equiv \lambda t.\lambda f.f$ discard an argument).[1] We now code terms as graphs—or *proofnets* in the terminology of linear logic. Represent application (@) and λ-abstraction by ternary nodes: the @-node has *continuation* and *argument ports*, which are called *auxiliary*, and a *function* port, which is called *principal.* A λ-node has auxiliary *body* and *parameter* ports, and a principal *root* port. Since a bound variable occurs exactly once, the wire

[1] Yet the language still has expressive power: for example, we could define True and False as $\lambda t.\lambda f.\langle t, f\rangle$ and $\lambda t.\lambda f.\langle f, t\rangle$, where $\langle x, y\rangle \equiv \lambda z.zxy$. Then let $\mathsf{Not} \equiv \lambda p.\lambda t.\lambda f.pft$ and $\mathsf{Or} \equiv \lambda p.\lambda q.\lambda t.\lambda f.p\,\mathsf{True}\,q(\lambda u.\lambda v.vIIIu)$. Add a fanout gate, and we can simulate circuits; thus evaluation is complete for PTIME.

coding its *occurrence* is connected to the λ-node coding its *binder.* A β-reduction happens when an edge connects an @- and λ-node by their principal ports: the graph is rewritten so that these nodes annihilate, and we fuse the wires previously connected with the @-continuation and the λ-body, and the λ-parameter and the @-argument (see Figure 1).

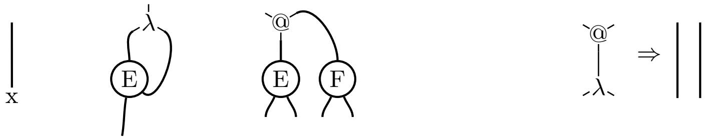

Fig. 1. Coding and reducing linear λ-terms as graphs.

Proposition 1. *Let G_Λ denote the graph coding of λ-term Λ. Then $E \to E'$ iff G_E reduces to $G_{E'}$ by the above annihilation and rewiring.*

Now interpret each graph node as a transformer on primitive *contexts* $c \in \Sigma^*$ (where $\Sigma = \{\circ, \bullet\}$, called *tokens*) which travel on graph wires. Entering an @-continuation or λ-body port transforms c to $\circ c$; entering an @-argument or λ-parameter port transforms c to $\bullet c$, and the context emerges at the respective principal port. Dually, entering an @-node (λ-node) at the principal @-function (λ-root) port with context τc, the token τ is popped, and the context c emerges at the @-continuation (λ-body) port if $\tau = \circ$, and at the @-argument (λ-parameter) port if $\tau = \bullet$. Notice that as context transformers on Σ^*, @- and λ-nodes are implemented by exactly the same hardware, which pushes and pops tokens, with routing to appropriate auxiliary ports in the latter case; we call this hardware a *fan node.*

The *ports* of a λ-term E (and its graph coding) are the dangling wire ends that represent the *root* and each free variable v of E; call these ports ρ and π_v. E has a *context semantics*—the relation given by paths between graph ports, where a path is defined by a *context* that is itself transformed by nodes along the path: $\mathcal{C}_E = \{(\langle c, \pi\rangle, \langle c', \pi'\rangle) \mid \text{context } c \text{ enters at port } \pi \text{ and exits as } c' \text{ at port } \pi\}$.

Proposition 2. *$\mathcal{C}_E$ is symmetric, and is preserved by β-reduction.*

Proof. The nodes forming a β-redex transform context c by pushing and then popping either $\circ$ (between an @-continuation and λ-body) or $\bullet$ (between an @-argument and λ-parameter), leaving c unchanged. The resulting paths are described by the wire fusings above.[2]

Linear λ-calculus is strongly normalizing, as reduction makes the λ-term (graph) get smaller. Since reduction preserves context semantics, the context semantics of a λ-term identifies its normal form. We now consider the process of *readback*: how can we reconstruct the normal form of E from $\mathcal{C}_E$?

[2] For this reason, the path algebra of Girard's GoI include equations like $p^*p+q^*q = 1$ (pushing and then popping $\circ$ [or $\bullet$] is the identity transformation), or $q^*p = 0$ (you cannot push $\circ$ and then pop $\bullet$).

Readback for linear λ-calculus. Let $N \equiv \lambda x_0.\cdots\lambda x_p.vN_0\cdots N_q$ be the normal form of E. Let $c_0 = \circ^+$ be some arbitrarily large string (stack) of $\circ$ tokens, enough so that, informally, as c_0 (or later, some variant of it) traces a path through the graph, the context is never empty when entering the principal port of a node, "stopping" the path. (We also write $\circ^k$ for the word consisting of k $\circ$ tokens.) If we add or subtract some small number of $\circ$ tokens to c_0, we don't care—we still write it as $\circ^+$. Insert c_0 at the root ρ: if $(\langle c_0, \rho\rangle), \langle c, \pi\rangle) \in \mathcal{C}_E$, what does $\langle c, \pi\rangle$ tell us about the normal form of E?

Proposition 3. *Context c identifies the* head variable *of N: $v = x_i$ iff $\langle c, \pi\rangle = \langle \circ^i \bullet \circ^+, \rho\rangle$, and v is a free variable if $\langle c, \pi\rangle = \langle \circ^+, \pi_v\rangle$.*

If $v = x_i$, then the *address* α of the head variable is $\circ^i\bullet$, otherwise the address is ϵ. Starting at port π above, $\alpha\,\circ^j\bullet$ defines a path ending at the root of subterm N_j. Recursing on this construction, we examine $\mathcal{C}_E$ for $(\langle \alpha\,\circ^j \bullet\circ^+, \pi\rangle, \langle c', \pi'\rangle)$; as in Proposition 3, c' identifies the head variable of $N_j \equiv \lambda y_0.\cdots\lambda y_r.wF_0\cdots F_s$: observe that $\langle c', \pi'\rangle$ is $\langle \circ^+, \pi_w\rangle$ if w is free, $\langle \circ^{i'} \bullet \circ^+, \rho\rangle$ if $w \equiv x_{i'}$, and finally $\langle \alpha\,\circ^j \bullet \circ^{j'} \bullet\circ^+, \pi\rangle$ if $w \equiv y_{j'}$. Thus contexts can be *decoded* to recover the head variables and subterms of all subexpressions of N.

Readback can be thought of as a *game* between an Opponent (Op) who wants to discover the normal form of a λ-term known by the Player (Pl). The *initial move* $\langle c_0, \rho\rangle$ codes the Op-question, "what is the head variable of the term?" The Pl-answer $\langle c, \pi\rangle$ is then *transformed* (by splicing $\circ^j\bullet$ into the context just before the $\circ^+$) into the next Op-question "what is head variable of the jth argument of the head variable?" and so on.[3] The Op-moves identify the addresses of subterms, the Pl-moves those of their head variables. A "winning strategy" for the Player means she can always respond to an Op-move; each λ-term codes a winning strategy for the Player—it gives her a λ-term to talk about. Games in the style of [1, 11] are essentially this guessing of normal forms. We use this *argot* to describe the input and output of contexts at graph ports.

How does the readback algorithm terminate? In linear λ-calculus, η-expansion preserves context semantics: the graph of $\lambda x.Ex$ pops and then pushes a token at its root. Thus there is no termination if the semantics is infinite: we just generate ever-larger pieces of the *Böhm tree*, a computation which quickly becomes boring. However, we can (nonconstructively) consider a *minimal* subset $\mathcal{C}^0 \subseteq \mathcal{C}$ where $(\langle c, \pi\rangle, \langle c', \pi'\rangle) \in \mathcal{C}^0$ iff for all $\sigma \in \Sigma^*$, $(\langle c\sigma, \pi\rangle, \langle c'\sigma, \pi'\rangle) \in \mathcal{C}$. Then readback terminates when all elements of $\mathcal{C}^0$ have been used.

What about terms not in normal form? In this case, readback constructs the normal form from contexts at the *ports*—but what do contexts mean as they travel through the interior of the graph? A "call" to a subgraph S, intuitively, requests the subterm of the graph that is bound to the head variable of the normal form of S. Consider a term $(\lambda x.E)F$; focus on the contexts that travel on the wire *down* from the @-node to the function $\lambda x.E$, and on the same wire *up* from the function to the @-node. First, a "call" is made *down* to the function; if

[3] The generation of Op-moves from Pl-answers is called *shunting* in [9], evoking a train at a head variable shunting off a "track" of applications to one of its arguments.

its head variable h_0 is the parameter, a context travels *up* that is the address $\bullet$ of the variable. Then the call is made to the argument F; if its head variable h_1, is bound, its address is routed back to h_0 to find the subterm (an argument of h_0) that is bound to h_1. In turn, the address of the head variable h_2 of that subterm may be routed back to h_1 in search of its bound subterm. And so on: head variable addresses in $\lambda x.E$ are used to find subterms in F and vice versa—like Player and Opponent, the two terms play against each other.

Finally, we should mention the connection to linear logic [7]. The graphs described are *proofnets* for *multiplicative linear logic*, which has logical connectives $\otimes$ (conjunction) and $⅋$ (disjunction). The $\otimes$ (alias @) *constructs a pair* from a continuation and an argument; the $⅋$ (alias λ) *unpairs* them for the body and parameter of the function. But if the nodes are logical connectives, what are the formulas? We discuss this briefly in Section 4.

3 Sharing

Now let's generalize linear λ-calculus, and allow multiple references to a bound variables. Three problems arise: What do we do when a bound variable is *not* referenced? How do we *combine* references? And *what* is being shared? To address these implementation issues, the graph technology, and its context semantics, get more complicated. First, some brief answers: In the case of no reference, we attach a unary *plug node* to the parameter: for example, in $\lambda x.\lambda y.x$, a plug is attached to the end of the wire coding y. Multiple references are combined with a fan node—the same kind of hardware used for @ and λ-nodes; we call it a *sharing node*. Expressions being shared by a bound variable are encapsulated in a *box*. To be used, a box needs to be *opened*; we introduce a binary *croissant node* that opens (erases) boxes. Boxed expressions may contain references to other boxed values; we thus introduce a binary *bracket node* to *absorb* one box into another, and a means to *fuse* one box with another. These ideas are now presented in the context of the call-by-name version of the λ-calculus.

Call-by-name coding and graph reduction. Figure 2 illustrates the coding; we write G_E for the coding of λ-term E. Observe that each free variable is represented by a single port. A *variable* v is coded as a croissant; when reduction attaches the wire for v to the boxed expression it is bound to, the croissant opens the box. An *abstraction* $\lambda x.E$ modifies G_E, using a λ-node to connect its root with the port of free variable x. An *application* EF is coded by coding E and F connected by an @-node, where F is additionally *boxed*.[4] The *principal port* of the box is located at the root of the graph coding F; each free variable v (coded by a croissant) in F, appearing as an *auxiliary port* of that box, is equipped with a bracket node that absorbs any box bound to v inside the box for F. Finally, if E and F share a free variable x, the two ports for x in their respective codings are combined with a sharing node.

We now describe three graph reduction schemes—*global*, *insular*, and *optimal*. Global and insular reductions are almost identical, and produce the same head

[4] The name CBN comes from the (en)*closure* of each argument by a box which can then be passed around without evaluating inside the box.

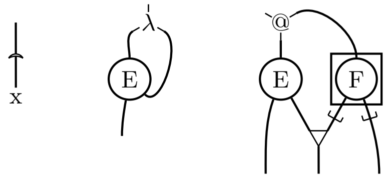

Fig. 2. Call-by-name graph coding.

normal forms. Insular and optimal reductions share the same context semantics on the *fragment* of the semantics used to compute readback. Combining these observations, we deduce that readback on optimally reduced graphs produces the head normal forms of global reduction—the crux of a correctness proof.

Global reduction (Figure 3) is the easiest to understand: $E \to_\beta E'$ iff $G_E \rhd_{\mathsf{G}} G_{E'}$. Define a *global box* to be a box together with the brackets *glued* to its free variable ports. A *global β-step* annihilates a λ-@ pair, and then propagates the (boxed) argument via *global duplications* and *global absorptions* to the variable occurrences, each a croissant that *globally opens* each propagated box.

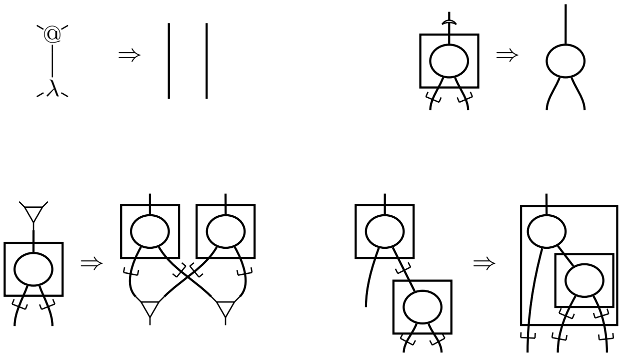

Fig. 3. Global reduction.

Insular reduction (Figure 4) resembles global reduction, except that brackets are detached from boxes, and croissants and brackets do not vanish when a box is opened. An *insular β-step* annihilates a λ-@ pair, and then pushes sharing, bracket, and croissant nodes as far as possible. Sharing nodes duplicate boxes, but not brackets; croissants and brackets open or add a box to an existing box, propagating to free variable ports; and two boxes adjacent (via a graph edge) can *fuse* along their common boundary.

Lemma 1. *Take a λ-term E and code it up as G_E. Perform leftmost-outermost* global *β-steps on G_E, deriving G'; also perform the same leftmost-outermost* insular *β-steps on G_E, deriving G''. Now erase all boxes, croissants, and brackets in G' and G''—they are now identical.*

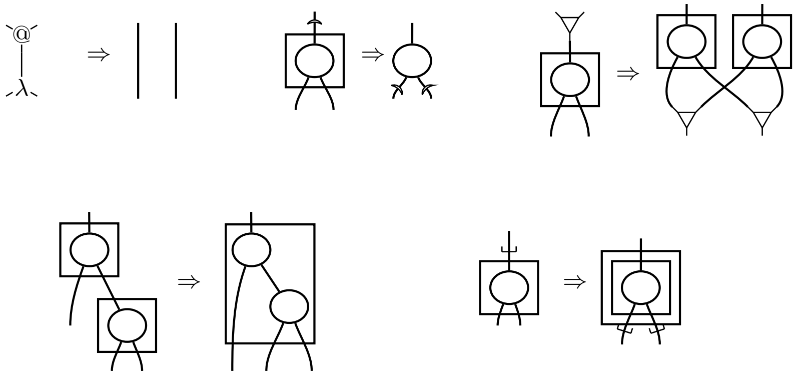

Fig. 4. Insular reduction.

Optimal reduction (Figure 5) looks totally different from global or insular reduction, since it has no global structure (boxes) in its reduction rules. (For those who object to globalization, here is the perfect interpreter: it is entirely local.) The initial coding G_E gives every graph node (@, λ, sharing, croissant, bracket) a *level*—the number of enclosing boxes. Then a croissant "opens" a boxed node by *decreasing* its level; a bracket "boxes" a node by *increasing* its level. In this way, the creation, opening, and sharing of boxes is done *incrementally.* Graphs quickly become inscrutable and do not look like λ-terms. But once we understand their *context semantics*, the resemblance to insular reduction will return.

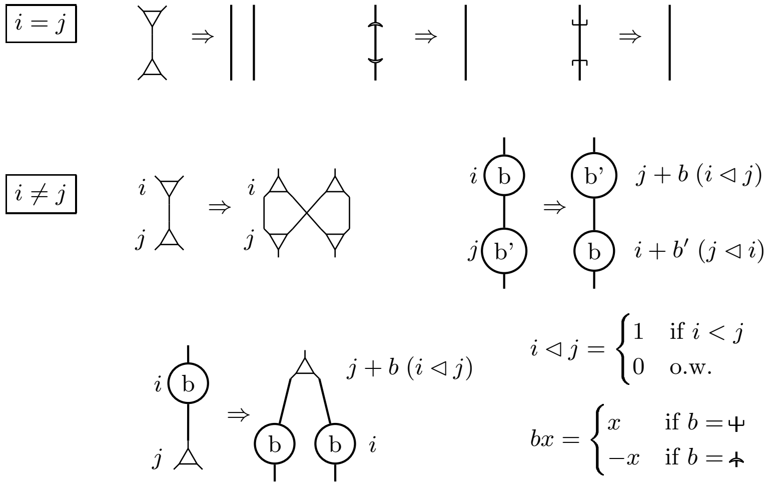
$i = j$
$i \neq j$
i
j
i
j
i
b
j
b'
b'
$j + b\ (i \lhd j)$
b
$i + b'\ (j \lhd i)$
i
b
j
$j + b\ (i \lhd j)$
b
b
i
$i \lhd j = \begin{cases} 1 & \text{if } i < j \\ 0 & \text{o.w.} \end{cases}$
$bx = \begin{cases} x & \text{if } b = \\ -x & \text{if } b = \end{cases}$

Fig. 5. Optimal graph reduction.

Context semantics for sharing. To implement a semantics for sharing, the *wires* of graphs for linear λ-calculus are generalized to *buses* of wires. A *context* is no longer a string, but a vector $\langle\!\langle \chi_1, \ldots, \chi_n, \circ^+ \rangle\!\rangle$ where datum χ_i travels on the ith wire of the bus. First we explain how contexts are modified by graph nodes. Then we explain what contexts mean—what they say about λ-terms, reduction, and sharing.

How: A graph node at *level* i modifies χ_i. The functions below respectively describe how a context is changed by a function (@ and λ), sharing, croissant, and bracket node, where the context is input to an *auxiliary* port of the graph node. For example, $\mathbf{f}_{i,\circ}$ pushes a $\circ$ token on the ith wire, describing the transformation of a context approaching the $\circ$ port of an @- or λ node. Sharing nodes are made from similar hardware: we push tokens L and R instead.[5] A croissant with level i, representing an occurrence of variable v, is interpreted semantically as $\mathbf{c}_{i,v}$, adding a "wire" to the context holding token v.[6] A bracket at level i is interpreted as $\mathbf{b}_i$, pairing two values onto one wire.

$$\begin{aligned}
\mathbf{f}_{i,t}\, \langle\!\langle \chi_1, \ldots, \chi_n, \circ^+ \rangle\!\rangle &= \langle\!\langle \chi_1, \ldots, \chi_{i-1}, t\chi_i, \chi_{i+1}, \ldots, \chi_n, \circ^+ \rangle\!\rangle \qquad [t \in \{\circ, \bullet\}]\\
\mathbf{s}_{i,t}\, \langle\!\langle \chi_1, \ldots, \chi_n, \circ^+ \rangle\!\rangle &= \langle\!\langle \chi_1, \ldots, \chi_{i-1}, t\chi_i, \chi_{i+1}, \ldots, \chi_n, \circ^+ \rangle\!\rangle \qquad [t \in \{\text{L}, \text{R}\}]\\
\mathbf{c}_{i,v}\, \langle\!\langle \chi_1, \ldots, \chi_n, \circ^+ \rangle\!\rangle &= \langle\!\langle \chi_1, \ldots, \chi_{i-1}, v, \chi_i, \ldots, \chi_n, \circ^+ \rangle\!\rangle\\
\mathbf{b}_{i}\, \langle\!\langle \chi_1, \ldots, \chi_n, \circ^+ \rangle\!\rangle &= \langle\!\langle \chi_1, \ldots, \chi_{i-1}, \langle \chi_i, \chi_{i+1} \rangle, \chi_{i+2}, \ldots, \chi_n, \circ^+ \rangle\!\rangle
\end{aligned}$$

The above functions *add* structure to a context; when a context encounters a croissant or bracket node at its *principal* port, structure is *consumed:*

$$\begin{aligned}
\mathbf{c}_{i,v}^{-1}\, \langle\!\langle \chi_1, \ldots, \chi_{i-1}, v, \chi_{i+1}, \ldots, \chi_n, \circ^+ \rangle\!\rangle &= \langle\!\langle \chi_1, \ldots, \chi_{i-1}, \chi_{i+1}, \ldots, \chi_n, \circ^+ \rangle\!\rangle\\
\mathbf{b}_{i}^{-1}\, \langle\!\langle \chi_1, \ldots, \chi_{i-1}, \langle \chi_i, \chi_{i+1} \rangle, \chi_{i+2}, \ldots, \chi_n, \circ^+ \rangle\!\rangle &=\\
&\langle\!\langle \chi_1, \ldots, \chi_{i-1}, \chi_i, \chi_{i+1}, \chi_{i+2}, \ldots, \chi_n, \circ^+ \rangle\!\rangle
\end{aligned}$$

At the principal port of a function (sharing) node, the path is routed to the left or right, in addition to the change of context—left if $t = \circ$ ($t = \text{L}$), and right if $t = \bullet$ ($t = \text{R}$), and token t is removed:

$$\begin{aligned}
\mathbf{f}_{i}^{-1}\, \langle\!\langle \chi_1, \ldots, \chi_{i-1}, t\chi_i, \chi_{i+1}, \ldots, \chi_n, \circ^+ \rangle\!\rangle &= \langle\!\langle \chi_1, \ldots, \chi_{i-1}, \chi_i, \chi_{i+1}, \ldots, \chi_n, \circ^+ \rangle\!\rangle\\
\mathbf{s}_{i}^{-1}\, \langle\!\langle \chi_1, \ldots, \chi_{i-1}, t\chi_i, \chi_{i+1}, \ldots, \chi_n, \circ^+ \rangle\!\rangle &= \langle\!\langle \chi_1, \ldots, \chi_{i-1}, \chi_i, \chi_{i+1}, \ldots, \chi_n, \circ^+ \rangle\!\rangle
\end{aligned}$$

Why: What does all this stuff *mean?* A context can now identify a head variable not only by its binders and arguments ("x, λ-bound at ..., which is the mth argument of head variable y, which is λ-bound at ..., which is the nth argument of head variable z, ...")—but also by its *occurrence*: which x is it? To provide this information, we need to know something about the *history of the computation* leading to the normal form. Context semantics provides this history.

[5] We could use $\circ$ and $\bullet$ instead of L and R, but we choose to increase "type safety" and make the contexts more readable.

[6] Again, v could be replaced with the null symbol ϵ, but we want to identify each croissant with the variable it represents.

Consider a context $\langle\!\langle \chi_1, \ldots, \chi_n, \circ^+ \rangle\!\rangle$ input at a port π of a graph.[7] If π is the root, each χ_{2i+1} traces a path along a chain of λ-binders, each χ_{2i} traces a path along a chain of applications, and if n is even (odd), it denotes an Opponent (Player) move that is input to (output from) the graph, identifying a subterm (head variable). Dually, when π is a free variable port, each χ_{2i+1} traces along applications, χ_{2i} along λ-binders, and and if n is odd (even), it denotes an Opponent (Player) move that is output from (input to) the graph. These parities change because at the root, the context enters with the orientation of a continuation, and at a free variable, with that of an expression.

Write each χ_i as $\overline{k}\sigma$, where $\overline{k}$ is a *numeral* $\circ^k\bullet$, and σ is a *sharing identifier.* The tokens from numerals are used by λ- and @-nodes, as in the linear case. Let V be a set of variables including those occurring in a λ-term; the sharing identifiers σ are generated from the grammar $\sigma \to V \mid \textsc{l}\sigma \mid \textsc{r}\langle\sigma, \sigma\rangle \mid \langle\sigma, \sigma\rangle$. Each sharing identifier σ explains how variable $\nu(\sigma)$ is shared, where $\nu(x) = x$, and $\nu(\textsc{l}\sigma) = \nu(\textsc{r}\langle\sigma', \sigma\rangle) = \nu(\langle\sigma', \sigma\rangle) = \nu(\sigma)$.

What do the sharing identifiers mean? A variable is the simplest identifier: in $\lambda x.x$, the initial Op-move $\langle\!\langle \circ^+ \rangle\!\rangle$ gives the Pl-response $\langle\!\langle \bullet x, \circ^+ \rangle\!\rangle$. In $\lambda x.xx$, the Player would respond $\langle\!\langle \bullet\textsc{l}x, \circ^+ \rangle\!\rangle$—$\textsc{l}x$ identifies the left occurrence. When the Opponent asks for the head variable of the argument of the head variable of $\lambda x.xx$, coded as the Op-move[8] $\langle\!\langle \bullet\textsc{l}x, \bullet\alpha, \circ^+ \rangle\!\rangle$, the Player responds with $\langle\!\langle \bullet\textsc{r}\langle\alpha, x\rangle, \circ^+ \rangle\!\rangle$— the *right* occurrence of x, occurring in a box that was "bound to" α. Finally, in $\lambda y.xy$, we have $\mathsf{Op}_1 = \langle\!\langle \circ^+ \rangle\!\rangle$ at the root, $\mathsf{Pl}_1 = \langle\!\langle x, \circ^+ \rangle\!\rangle$ at π_x, $\mathsf{Op}_2 = \langle\!\langle x, \bullet\alpha, \circ^+ \rangle\!\rangle$ at π_x, and $\mathsf{Pl}_2 = \langle\!\langle \bullet\langle\alpha, y\rangle, \circ^+ \rangle\!\rangle$ at the root—the sharing identifier $\langle\alpha, y\rangle$ says y is in a box "bound to" α, but not shared. In summary: σ represents the sharing of $\nu(\sigma)$. When $\sigma = \langle\sigma', \sigma''\rangle$ or $\textsc{r}\langle\sigma', \sigma''\rangle$, the σ' describes the *external* sharing of the occurrence of the box containing the occurrence of $\nu(\sigma)$, and σ'' describes the *internal* sharing of $\nu(\sigma)$ inside the box. Check your intuitions with the following examples of contexts input and output (all at the root) to simple λ-terms; since all contexts have the form $\langle\!\langle \ldots, \circ^+ \rangle\!\rangle$—where the $\circ^+$ represents a "call" to a subterm—we henceforth eliminate its mention.

$\underline{\lambda x.\lambda y.xy}$**:** Then $\mathsf{Op}_1 = \langle\!\langle\ \rangle\!\rangle$, $\mathsf{Pl}_1 = \langle\!\langle \bullet x \rangle\!\rangle$ ("the head variable is x, bound"), $\mathsf{Op}_2 = \langle\!\langle \bullet x, \bullet\alpha \rangle\!\rangle$ ("the first argument of x—which the Opponent 'binds' to α—what's its head variable?"), $\mathsf{Pl}_2 = \langle\!\langle \circ \bullet \langle\alpha, y\rangle \rangle\!\rangle$ ("y, free in a box bound to α").

$\underline{\lambda x.x(\lambda y.y)}$**:** Again $\mathsf{Op}_1 = \langle\!\langle\ \rangle\!\rangle$, $\mathsf{Pl}_1 = \langle\!\langle \bullet x \rangle\!\rangle$ and $\mathsf{Op}_2 = \langle\!\langle \bullet x, \bullet\alpha \rangle\!\rangle$; but now $\mathsf{Pl}_2 = \langle\!\langle \bullet x, \bullet\alpha, \bullet y \rangle\!\rangle$ (contrast with the previous example).

$\underline{\lambda x.x(\lambda y.xy)}$**:** $\mathsf{Op}_1 = \langle\!\langle\ \rangle\!\rangle$, $\mathsf{Pl}_1 = \langle\!\langle \bullet\textsc{l}x \rangle\!\rangle$, $\mathsf{Op}_2 = \langle\!\langle \bullet\textsc{l}x, \bullet\alpha_0 \rangle\!\rangle$, $\mathsf{Pl}_2 = \langle\!\langle \bullet\textsc{r}\langle\alpha_0, x\rangle \rangle\!\rangle$, $\mathsf{Op}_3 = \langle\!\langle \bullet\textsc{r}\langle\alpha_0, x\rangle, \bullet\alpha_1 \rangle\!\rangle$, $\mathsf{Pl}_3 = \langle\!\langle \bullet\textsc{l}x, \bullet\alpha_0, \bullet\langle\alpha_1, y\rangle \rangle\!\rangle$.

$\underline{\lambda x.\lambda y.\lambda z.\lambda w.x(y(zw))}$**:** $\mathsf{Op}_1 = \langle\!\langle\ \rangle\!\rangle$, $\mathsf{Pl}_1 = \langle\!\langle \bullet x \rangle\!\rangle$, $\mathsf{Op}_2 = \langle\!\langle \bullet x, \bullet\alpha_1 \rangle\!\rangle$, $\mathsf{Pl}_2 = \langle\!\langle \circ \bullet \langle\alpha_1, y\rangle \rangle\!\rangle$, $\mathsf{Op}_3 = \langle\!\langle \circ \bullet \langle\alpha_1, y\rangle, \bullet\alpha_2 \rangle\!\rangle$, $\mathsf{Pl}_3 = \langle\!\langle \circ \circ \bullet\langle\alpha_1, \langle\alpha_2, z\rangle\rangle \rangle\!\rangle$, $\mathsf{Op}_4 = \langle\!\langle \circ \circ \bullet\langle\alpha_1, \langle\alpha_2, z\rangle\rangle, \bullet\alpha_3 \rangle\!\rangle$, $\mathsf{Pl}_4 = \langle\!\langle \circ \circ \circ \bullet \langle\alpha_1, \langle\alpha_2, \langle\alpha_3, w\rangle\rangle\rangle \rangle\!\rangle$.

[7] For now, imagine the graph is of a normal λ-term.

[8] Literally, $\alpha \equiv \epsilon$, but I include this notation to identify the Opponent's move in the context. Think of the Opponent as a projection function with head variable identified by α.

Exercise: reading back the normal form of a normal λ-term. Given a *closed* λ-term $N \equiv \lambda x_0.\cdots\lambda x_p.x_i N_0 \cdots N_q$ in *normal form,* how do we read back N from the context semantics of G_N? We mimic the procedure introduced for linear λ-calculus, using the refined idea of contexts described above. Recall Op-moves identify subterms, and Pl-moves identify head variables; an Op-move has the form $O = \langle\!\langle \ell_1\sigma_1, a_1\alpha_1, \ldots, \ell_n\sigma_n, a_n\alpha_n \rangle\!\rangle$, and a Pl-move has the form $P = \langle\!\langle \ell_1\sigma_1, a_1\alpha_1, \ldots, \ell_n\sigma_n, a_n\alpha_n, \ell_{n+1}\sigma_{n+1} \rangle\!\rangle$, where the ℓ_i and a_i are numerals. The Opponent generates new moves by taking a Pl-move (coding a head variable v), and adding $\overline{k}\alpha$ just before the $\circ^+$ component ("what is the head variable of the kth argument of v?"). The α marks the occurrence of a projection variable from the Opponent.

Follow the path from the root to a subterm defined by a context: $\ell_1 = \circ^i\bullet$ traces through the λ-binders at the top level, reaching the head variable x_i of N. Then σ_1 represents *some* occurrence of x_i, and that occurrence (marked in the graph by a croissant node) is at *level* $d = \partial(\sigma_1)$, where $\partial(x) = 0$, $\partial(\alpha_j) = 1$, $\partial(\textsc{l}\sigma) = \partial(\sigma)$, and $\partial(\textsc{r}\langle\sigma', \sigma\rangle) = \partial(\langle\sigma', \sigma\rangle) = \partial(\sigma') + \partial(\sigma)$. The path then unpacks $\alpha'_1, \ldots, \alpha'_d$ from σ_1, and consumes the token x_i. Next $a_1 = \circ^j\bullet$ is consumed to trace through a chain of @-nodes, en route to the graph of subterm N_j, while placing α_2 from the context on the $(d+1)$st wire. The explanation continues inductively on $\langle\!\langle \ell_2\sigma_2, a_2\alpha_2, \ldots, \ell_n\sigma_n, a_n\alpha_n \rangle\!\rangle$.

For the head variable identified by the Player in move P above, $\ell_{n+1}\sigma_{n+1}$ gives the numeral ℓ_{n+1} coding the lexical address of the head variable $v = \nu(\sigma_{n+1})$ in the subterm N' containing v's *binder*, and $\partial(\sigma_{n+1})$ gives the *distance* (number of enclosing boxes) separating that binder from the *occurrence* of v in the enclosed subterm N''. The remaining information in move P gives the address of N'. Moreover, σ_{n+1} identifies the occurrences of the Op-variables bound to each of the boxes separating the occurrence of v from its binder.

Readback does not require any information from the σ_i, *except* that $\partial(\sigma_i)$ be correct, since it indicates how many boxes have to be entered to get to the head variable in question. The rest of the information in σ_i tells what variable occurrences $\boldsymbol{v}$ were bound to these boxes during reduction, and what variable occurrences $\boldsymbol{v'}$ were bound to boxes containing $\boldsymbol{v}$, and so on. Now consider readback on a term that is strongly normalizable: use *insular reduction* to compute its normal form.

Proposition 4. *In an insular β-step, when a variable x is bound to a box* B*, the sharing, bracket, and—in particular—croissant nodes at the* occurrences *of x move to the free variable ports of the copies of* B *produced during the reduction. In that latter position, they* do not change the level *of head variables; equivalently, their contribution to sharing identifiers σ in the context semantics code the correct depth.*

As a consequence, when entering the principal port of a box, the context resembles that which is found in readback of a normal λ-term.

Proposition 5. *When the principal port of a box at level d is encountered on a path, the first d wires of the context hold $\alpha'_1, \cdots \alpha'_d$, all from projection functions of the Opponent.*

In other words: paths from head variables to the root pack sharing information into the context, which is *unpacked* (in the same graph location!) on return paths to subterms. It does not matter *what* this information is, as long as it has the correct depth. Recall Lemma 1, and let G' and G'' be the respective graphs derived by global and insular reduction to the normal form of the same λ-term: then G', G'' only differ in the accumulation of bracket and croissant nodes at free variable ports of boxes. In G'', these *control nodes* are the debris of β-contraction—but for mere readback of the λ-term, they are only sound and fury, signifying nothing.

Lemma 2. *Let G_E be the graph coding a λ-term E. Given the context semantics of the* normalized *graph produced by either global or insular reduction of G_E, the readback algorithm computes the normal form of E.*

Readback: the general case and optimal reduction. However, both global and insular reduction can *change* the context semantics. For example, in $(\lambda z.\lambda w.wzz)(xy)$, the context semantics includes paths from π_x to π_y that do not include sharing information (i.e., the use of L or R). But in the context semantics of global or insular normal forms, from which we can read back $xy(xy)$, this information is required, since a sharing node is found on every path from π_x to π_y. So if the context semantics changes, what relation is there between the context semantics of the graph at the start and conclusion of reduction? How do we know that we are reading back the normal form of the right term? The solution is to recognize that insular reduction does not change the fragment of the context semantics that is used for readback. Readback traces paths in a *proper* manner, a straightforward idea that we now elaborate:

Definition 1. *A* proper interaction *with a box H at level k is a sequence of contexts $I = \{o_1, p_1, \ldots, o_n, p_n\}$ that alternately enter and exit the ports of H, where*

1. *Let $1 \leq r \leq k$; the rth component of every context in I is a sharing identifier. Moreover, any two contexts in I have the same such identifier in the rth component.*
2. *They are consistent with the context semantics: the input of o_i followed by the output of p_i is determined by the functions defined by nodes along the path from o_i to p_i;*
3. *o_1 is input at the principal port (root) of the box, and each o_{i+1} is input at the port where p_i was output;*
4. *If the port of p_i and o_{i+1} is not the root, then their $(k+1)$st wires contain the same sharing identifier (i.e., they are talking about the same variable occurrence in the box); and*
5. *Any interactions with boxes contained in H are also proper.*

Lemma 3. *The context semantics defines an interaction with an implicit "box" that encloses the entire graph. In readback on an insular or global normal form, this interaction is proper. Furthermore, an interaction with a box is proper iff an interaction with any* insular *graph reduction of that box is proper. (The latter assertion is false for global reduction.)*

The proof of this lemma is a slightly tedious but straightforward induction on reduction sequences, where we check that each insular graph reduction rule preserves the requisite properties. Since readback on an (insular) normalized graph is a proper interaction, we conclude:

Theorem 1. *Let G_E be the graph coding a λ-term E. Given the context semantics for G_E, the readback algorithm produces the normal form of E. Since optimal reduction leaves the* entire *context semantics invariant, readback on optimally reduced graphs is correct.*

We remark that in readback of an optimally reduced graph, we use the graph to *compute* the necessary fragment of the context semantics, rather than being presented with it extensionally. Observe that this theorem can be strengthened so that E need not have a normal form; in this case, the readback algorithm computes the Böhm tree.

4 Types

We can *type* terms by attaching data types that are linear logic formulas to oriented graph edges. A λ-node (@-node) has an outgoing (ingoing) type $\alpha \multimap \beta$ on its principal root (function) port, an outgoing (ingoing) type α on its auxiliary parameter (argument) port, and an ingoing (outgoing) type β on its auxiliary body (continuation) port. A box has ingoing types $!\alpha_1, \ldots, !\alpha_n$ on its auxiliary ports, and an outgoing type $!\beta$ on its principal port. A sharing node has $!\alpha$ going in at its principal port, and $!\alpha$ going out at its auxiliary ports. A croissant (bracket) has $!\alpha$ ($!\alpha$) going in at its principal port, and α ($!!\alpha$) at its auxiliary port. The CBN coding assumes arguments are always boxed, so simple types τ are compiled into linear logic types $[\tau]$ as $[A] = A$ and $[\alpha \rightarrow \beta] = ![\alpha] \multimap [\beta]$. These types are preserved by global and insular reduction, but not by optimal reduction.[9]

In the bus system, a subterm of type $! \cdots !\alpha$ (m !s) must be enclosed in m boxes, and each box has a dedicated wire on the bus, holding information about the *sharing* of that box—what variable occurrence is bound to it? From these contexts we can read part of the computation history. The information flow on each wire clarifies how context semantics can be interpreted as a kind of flow analysis, describing the fine structure of games in the style of McCusker's informal presentation [2]. The Curry-Howard correspondence is reinterpreted using the games slogan: types are *arenas* where games can be played (Player: "I am thinking of a term of type τ"; Opponent: "What is its head variable?"...), and a λ-term of that type represents a winning Pl-strategy.[10] In the typed framework we can dispense with the $\circ^+$-notation, using the types to insert the exact number of needed tokens. For example, the Pl-strategy 3 has $\mathsf{Op}_1 = ?$,

[9] Consider the sharing of a function with type $!\mathsf{Int} \multimap \mathsf{Int}$. By duplicating the λ-node, we then need to share the output of type Int—without a !, which is type incorrect.

[10] In game semantics, the *moves* of game $A \otimes B$ are defined as the disjoint union of A and B; the $\bullet$ and $\circ$ tokens of context semantics are what implement this disjoint union. In this spirit, context semantics realizes a sort of "machine language" for implementing games.

$\mathsf{Pl}_1 = 3$; the strategy $\mathsf{succ} : \mathsf{Int} \multimap \mathsf{Int}$ has (for each n) $\mathsf{Op}_1 = \circ?$, $\mathsf{Pl}_1 = \bullet?$, $\mathsf{Op}_2 = \bullet n$, $\mathsf{Pl}_2 = \circ(n+1)$; and $\mathsf{succ}' : !\mathsf{Int} \multimap \mathsf{Int}$ has $\mathsf{Op}_1 = \langle\!\langle \circ? \rangle\!\rangle$, $\mathsf{Pl}_1 = \langle\!\langle \bullet\alpha, ? \rangle\!\rangle$, $\mathsf{Op}_2 = \langle\!\langle \bullet\alpha, n \rangle\!\rangle$, $\mathsf{Pl}_2 = \langle\!\langle \circ(n+1) \rangle\!\rangle$. Observe that a wire of type Int carries requests (of type $\mathsf{Int}^{\perp}$) for an integer data in one direction, and supplies an integer (Int) in the other direction. An edge of type $!\mathsf{Int}$ also has a dedicated "sharing wire" identifying which shareholder wants the data; an edge of type $!!\mathsf{Int}$ also explains how the shareholder is also shared. A free variable $x : !\mathsf{Int}$ occurring in a box has an *internal* sharing wire (how is x shared in the box?), and an *external* sharing wire (how is the box shared?) that is "threaded" through the entire box. Upon exiting at x's port, these wires are paired by the bracket. When a !!-type becomes a !-type inside a box without transformation by any graph node, only the type information internal to the box is being shown.

Consider the example $(\lambda f : !(!\mathsf{Int} \multimap \mathsf{Int}).f(f(!3)))\mathsf{succ}' : \mathsf{Int}$ (annotated using the CBN translation), where $\mathsf{Op}_1 = ?$ and $\mathsf{Pl}_1 = 5$—but what are the calls to the *boxed* version of succ' during the computation? The first call is $\mathsf{C}_1 = \langle\!\langle \textsc{l}f, \circ? \rangle\!\rangle$ (what's the output?), and succ' responds $\mathsf{S}_1 = \langle\!\langle \textsc{l}f, \bullet\alpha, ? \rangle\!\rangle$ (what's the input?); then $\mathsf{C}_2 = \langle\!\langle \textsc{r}\langle \alpha, f \rangle, \circ? \rangle\!\rangle$ (what's the output?—but for a *new* call site), $\mathsf{S}_2 = \langle\!\langle \textsc{r}\langle \alpha, f \rangle, \bullet\alpha, ? \rangle\!\rangle$ (again, what's the input?), $\mathsf{C}_3 = \langle\!\langle \textsc{r}\langle \alpha, f \rangle, \bullet\alpha, 3 \rangle\!\rangle$ (the input is 3), $\mathsf{S}_3 = \langle\!\langle \textsc{r}\langle \alpha, f \rangle, \circ 4 \rangle\!\rangle$ (the output is 4), $\mathsf{C}_4 = \langle\!\langle \textsc{l}f, \bullet\alpha, 4 \rangle\!\rangle$ (the input is 4), $\mathsf{S}_4 = \langle\!\langle \textsc{l}f, \circ 5 \rangle\!\rangle$ (the output is 5). Each wire holds type-dedicated information; for example, in S_3, $\textsc{r}\langle \alpha, f \rangle$ has type $!(!\mathsf{Int} \multimap \mathsf{Int})$ (sharing the function), and $\circ 4$ has type $!\mathsf{Int} \multimap \mathsf{Int}$; the tag $\circ$ indicates an injection from Int. Dually, in C_4, $\textsc{l}f : [!(!\mathsf{Int} \multimap \mathsf{Int})]^{\perp}$, $\bullet\alpha : (!\mathsf{Int} \multimap \mathsf{Int})^{\perp} = !\mathsf{Int} \otimes \mathsf{Int}^{\perp}$ (where $\bullet$ is the injection from $!\mathsf{Int}$), and $3 : (\mathsf{Int} \multimap \perp)^{\perp} = \mathsf{Int}$. In an initial coding, a bus for an edge of type α has wires of type $\alpha, \phi(\alpha), \phi^2(\alpha), \ldots$, where $\phi^{n+1}(\alpha) = \perp$ for a base type, $\phi^{n+1}(!\alpha) = \phi^n(\alpha)$, and $\phi^{n+1}(\alpha \multimap \beta) = \phi^n(\alpha) \multimap \phi^n(\beta)$. This elaborated typing can be shown to be preserved by optimal reduction.

5 Labelled $\boldsymbol{\lambda}$-calculus *à la Lévy* and paths

Give every application and abstraction a unique *atomic* label, and define *labelled β-reduction* as $(\lambda x.E)^{\ell} F \rhd_{\mathsf{lab}} ([F^{\underline{\ell}}/x]E)^{\ell}$. For example, consider *labelled reduction* of $(\lambda x.xx)(\lambda x.xx)$ (see Figure 6):

$$((\lambda x.(x^1 x^2)^3)^4 (\lambda x (x^5 x^6)^7)^8)^9 \rhd_{\mathsf{lab}} ((\lambda x.(x^5 x^6)^7)^{8\underline{4}1} (\lambda x (x^5 x^6)^7)^{8\underline{4}2})^{349}$$
$$\rhd_{\mathsf{lab}} ((\lambda x.(x^5 x^6)^7)^{8\underline{4}2\underline{8\underline{4}1}5} (\lambda x.(x^5 x^6)^7)^{8\underline{4}2\underline{8\underline{4}1}6})^{78\underline{4}1349}$$

Now interpret underlining as *reversal*, so $\underline{\underline{\ell}} = \ell$ and $\underline{\ell\ell'} = \underline{\ell'}\,\underline{\ell}$. For example, we have $8\underline{4}2\underline{8\underline{4}1}5 = 8\underline{4}2\underline{1}4\underline{8}5$. Observe that this label on the function in the last application above, describes a path between an @- and λ-node in the initial graph coding of $(\lambda x.xx)(\lambda x.xx)$—read underlined atomic labels by *reversing* the orientation of the graph edge. This connection between labels and paths has been studied in [5]; we add some intuitions. First, except for K-redexes, which discard term structure, the history of the computation is coded in the labels: we can effectively run the reductions backwards. But the same is not true of context semantics—we cannot reconstruct the initial term. For example, take $((\lambda x.(x^1 x^2)^3)^4 (\lambda y.y^5)^6)^7$, which reduces to $(\lambda y.y^5)^{\ell}$ where $\ell = 6\underline{4}2\underline{6\underline{4}1}56\underline{4}1347$—try tracing this path, for fun. But the context semantics of this term is just that

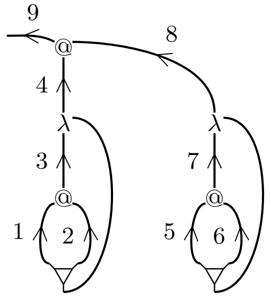

Fig. 6. Labelled reduction and paths.

of $\lambda y.y$—so what is it about reductions that lets ℓ get *forgotten* by the context semantics?

Just as optimal reductions commute bracket and croissant nodes through @ and λ-nodes, rewrite $(U^\alpha V)$ as $(UV^{\underline{\alpha}})^\alpha$ and $(\lambda x.E)^\alpha$ as $\lambda x.E^\alpha[x^{\underline{\alpha}}/x]$. Forget all atomic labels except those marking variable occurrences. Then the labelling of $(\lambda y.y^5)^\ell$ gets rewritten to $(\lambda y.y^5)^{2\underline{1}51}$ and again to $\lambda y.y^{\underline{2\underline{1}51}52\underline{1}51}$. By adding rules that commute and annihilate atomic labels, imitating those for optimal reduction, we should be able to deduce that the outer label self-annihilates.

6 Conclusions and open problems

This largely tutorial paper represents an important part of research: to *re-search* some of what has already been found, and to know it better. Context semantics is an expressive foundation for static program analysis. We have given a simple correctness proof, explaining why *readback* of the semantics—unchanged by optimal reduction—yields the normal form. What *fragments* of this semantics correspond to *tractable* schemes for static program analysis? It would be nice to tie this semantics more closely to that of games, so that an Opponent could ask questions about aspects of the computation history, i.e., "what variable got bound to the box containing the head variable?". These sorts of questions suggest the possibility of full abstraction theorems that are sensitive to *sharing*—that two $\beta\eta$-equivalent terms are distinguishable if they share applications differently. The informal typing of individual wires in Section 4 could be a step in providing a semantics of optimal reduction where the nodes in "negative" position are explained properly. The relation of labels to semantics, briefly discussed in Section 5, deserves a clearer and more detailed explanation. The box-croissant-bracket metaphor, as explained in [3], is just a *comonad*: the box is a functor, and the croissant and bracket are the *unit* and *bind* described in [14]. Why not use graph reduction, then, as a means of implementing monads? For example, *state* can be represented by the !-dual functor ?, so in a game for ?α, the Opponent could query the state as well as the type α. There is more of this story to be told.

Dedicated in memory of my father, Theodore Mairson (1919–2002)

Acknowledgements. Thanks to many people I have discussed these ideas with over several years, including especially Alan Bawden, Jakov Kucan, Julia Lawall, Jean-Jacques Lévy, Peter Neergaard, and Luca Roversi. Special thanks to Peter who did the figures for me at the last minute.

References

1. S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, Dec. 2000.
2. S. Abramsky and G. McCusker. Game semantics. In H. Schwichtenberg and U. Berger, editors, *Theoretical Foundations of Computer Graphics and CAD*, volume 165 of *NATO ASI*, pages 307–325. Springer-Verlag, 1999.
3. A. Asperti. Linear logic, comonads and optimal reductions. *Fundamentae Informaticae*, 22:3–22, 1995.
4. A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages.* Cambridge University Press, 1998.
5. A. Asperti and C. Laneve. Paths, computations and labels in the λ-calculus. *Theoretical Computer Science*, 142(2):277–297, 15 May 1995.
6. A. Asperti and H. G. Mairson. Parallel beta reduction is not elementary recursive. *Information and Computation*, 170:49–80, 2001.
7. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50, 1987.
8. J.-Y. Girard. Geometry of interaction I: Interpretation of system F. In C. Bonotto, R. Ferro, S. Valentini, and A. Zanardo, editors, *Logic Colloquium '88*, pages 221–260. North-Holland, 1989.
9. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992*, pages 15–26, New York, NY, USA, 1992. ACM Press.
10. G. Gonthier, M. Abadi, and J.-J. Lévy. Linear logic without boxes. In *Proceedings 7th Annual IEEE Symp. on Logic in Computer Science, LICS'92, Santa Cruz, CA, USA, 22–25 June 1992*, pages 223–34. IEEE Computer Society Press, Los Alamitos, CA, 1992.
11. J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Information and Computation*, 163(2):285–408, Dec. 2000.
12. J. Lamping. An algorithm for optimal lambda calculus reduction. In *POPL '90. Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, January 17–19, 1990, San Francisco, CA*, pages 16–30, New York, NY, USA, 1990. ACM Press.
13. J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul.* PhD thesis, Université Paris 7, 1978. Thèse d'Etat.
14. P. Wadler. The essence of functional programming. In *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992*, pages 1–14, New York, NY, USA, 1992. ACM Press.