Sharing continuations: proofnets for languages with explicit control

Julia L. Lawall*
Harry G. Mairson**

Department of Computer Science
111 Cummington Street
Boston University
Boston, Massachusetts 02215
{jll,mairson}@cs.bu.edu

Abstract. We introduce graph reduction technology that implements functional languages with control, such as Scheme with `call/cc`, where continuations can be manipulated explicitly as values, and can be optimally reduced in the sense of Lévy. The technology is founded on *proofnets* for multiplicative-exponential linear logic, extending the techniques originally proposed by Lamping, where we adapt the continuation-passing style transformation to yield a new understanding of sharable values. Confluence is maintained by returning multiple answers to a (shared) continuation.

Proofnets provide a *concurrent* version of linear logic proofs, eliminating structurally irrelevant sequentialization, and ignoring asymmetric distinctions between inputs and outputs—dually, expressions and continuations. While Lamping's graphs and their variants encode an embedding of intuitionistic logic into linear logic, our construction implicitly contains an embedding of classical logic into linear logic.

We propose a family of translations, produced uniformly by beginning with a continuation-passing style semantics for the languages, employing standard codings into proofnets using call-by-value, call-by-name—or hybrids of the two—to locate proofnet *boxes*, and converting the proofnets to direct style. The resulting graphs can be reduced simply (cut elimination for linear logic), have a consistent semantics that is preserved by reduction (geometry of interaction, via the so-called *context semantics*), and allow shared, incremental evaluation of continuations (optimal reduction).

1 Introduction

Expressions and continuations are dual, separate but equal computational structures in a programming language. The former provides a value; the latter con-

* Supported by NSF Grant EIA-9806718.
** Supported by NSF Grants CCR-9619638 and EIA-9806718, and the Tyson Foundation.

sumes it. Yet evaluating expressions is very familiar, while evaluating continuations is considered esoteric, even though both are made of the same stuff. The incorporation of continuations as first-class citizens in programming languages was not welcomed like the Emancipation Proclamation, but instead regarded warily as a kind of witchcraft, with implementation pragmatics that are ill-defined and unclear. If expressions and continuations are indeed dual, then so should be the technology of their implementation, and the flexibility with which we reason about them. Efficient evaluation of one should reveal dual strategies for evaluating the other. In short, everything we know about expressions we ought to know about continuations.

We take a significant step towards this equality by formulating a general version of graph reduction that implements the sharing and optimal incremental evaluation of both expressions and continuations, each evaluated using the same primitive operations. By founding our technology on generic tools from logic and programming language theory, specifically the CPS transform and its relation to linear logic, we are for the first time able to produce a family of related implementations in an entirely mechanical way.

Nishizaki earlier produced a coding of Scheme with `call/cc` in linear logic, via *ad hoc* reasoning, based on a proof of a proposition of linear logic corresponding to the type of `call/cc` [22]. In contrast, our new contribution is to produce Nishizaki's coding, and many others, by a mechanical process based on the denotational semantics of the programming language. Not only do we get a much deeper insight into principles, we greatly simplify the problem of constructing graph reduction implementations of other languages with explicit control. In bringing ideas from logical theory closer to implementation technology, we hope to make researchers think about the pragmatics of continuations in simple, novel, and useful ways.

Our methodology is founded on proofnets for multiplicative-exponential linear logic, following the beautiful insights of John Lamping [17], who realized Jean-Jacques Lévy's specification of correct, optimal reduction for the λ-calculus [18], and of Gonthier, Abadi, and Lévy, who reinterpreted Lamping's insights in the guise of Girard's geometry of interaction, and the related embedding of intuitionistic logic in linear logic [12, 13]. Linear logic [11] provides an ideal substrate for the implementation of control operators, as it makes no asymmetric distinctions between inputs and outputs, or analogous expressions and continuations. We extend the optimal reduction technology to implement explicit control and sharing of continuations, essentially via an embedding of *classical* logic in linear logic, following a line of research beginning with Griffin and then Murthy [14, 21]. Our construction is based on continuation-passing style, but generates direct-style graphs. This approach extends to implement most any functional language with abortive control operators whose semantics can be described in continuation-passing style—for example, Filinski's *symmetric λ-calculus* [8], and Parigot's *λ_μ-calculus* [23].[1]

[1] We have implemented these languages using the techniques in this paper. This work will be included in a later, extended version of the manuscript.

What do optimality and correctness mean in a language with explicit control? To understand *optimality* and *sharing* in the context of continuations, consider the evaluation of a Scheme expression

```
([fun₁]
   (call/cc (lambda (a) ([fun₂]
      (call/cc (lambda (b) [exp]))))))
```

in the context of some complex continuation k. The expression $[exp]$ can *implicitly* access its current continuation c, or *explicitly* access continuations a and b named by `a` and `b`. All of these continuations extend the continuation of the entire expression k. Optimality ensures that a, b, and c share k, that b and c share a, and so on. If continuations are shared, duplicate work can be avoided as continuations are simplified.

A reduction strategy ρ is *correct* if for any expression E, if there is some strategy σ that reduces E to a normal form, then ρ also reduces E to a normal form. In the absence of control operators, normal forms in the λ-calculus are unique, so all correct strategies produce the same normal form, if one exists. However, control operators destroy the uniqueness of normal forms: let E be the Scheme expression `(call/cc (lambda (k) ((lambda (x) 1) (k 2))))`; a call-by-name strategy reduces E to 1, while a call-by-value strategy reduces E to 2. A correct evaluation strategy cannot simply choose one of these answers. Define context C as `(if (= [−] 1) 0 ⊥)` and C' as `(if (= [−] 1) ⊥ 0)`. Since $C[E]$ evaluates to 0 under call-by-name and diverges under call-by-value, a correct evaluation of E must return 1. But since $C'[E]$ evaluates to 0 under call-by-value and diverges under call-by-name, a correct evaluation of E must return 2. Returning both 1 and 2 in the evaluation of E is *not* contradictory: it merely amounts to supplying both answers to a single *shared* continuation.

Technical contributions: The efficiency of optimal reduction is based on the incremental propagation of sharing nodes. Implementations of optimal reduction based on linear logic, as proposed by Gonthier, Abadi, and Lévy, and later by Asperti, use proofnet *boxes* to coordinate these interactions. Nevertheless, the boxing strategy only permits the sharing of values. To extend this technology to languages with control operators, the key technical question is: where do we put the boxes to allow the sharing of continuations?

Our solution exploits the *continuation-passing style (CPS) transformation.* If a language with control operators can be translated into the pure λ-calculus using a CPS transformation, we can use existing technology to construct the graph of the CPS transformed term. The CPS translation of a term is more verbose than the original term, and more expensive to reduce to a normal form. We show that for a language with abortive continuations, the graphs of CPS terms can be mechanically converted back to direct style, maintaining the boxing of continuations induced by the CPS term. The transformation "rotates" principal ports of boxes so that continuations can be copied. We prove that this transformation does not change the underlying denotational semantics of the terms, as defined by the geometry of interaction. This approach can be applied to any variant of the CPS transformation, and any strategy for coding pure λ-terms as proofnets. The result is a family of possible translations into graphs, and these graphs can be optimally reduced in the sense of Lévy's labelled terms.

Traditionally, compiler optimizations have addressed sharing of expressions. The technology presented here provides a new systematic basis on which to optimize the sharing of continuations.

In summary, all of the translations we outline possess a simple graph reduction on translated terms (cut elimination for linear logic), a consistent semantics that is preserved by reduction (geometry of interaction, via the so-called *context semantics* of Gonthier [12]), and a mechanism whereby continuations can be incrementally evaluated (optimal reduction). The situation of this technology within multiplicative-exponential linear logic ensures that the semantic characterization given is equivalent to the operational semantics of graph reduction. Viewing data types as games, and contexts (in the sense of Gonthier) as moves in a composite game, one immediately suspects that categories of games should provide the right kind of "more abstract" semantics for calculi with explicit control. Furthermore, full abstraction theorems for languages with control seem to be easily accessible, given the full completeness results for linear logic.

2 Preliminaries

We briefly sketch the construction of graphs to implement λ-calculus; more details can be found elsewhere [1, 12]. Graphs are composed of wires and fixed-arity nodes, as well as *boxes*, which enclose subgraphs. The λ-calculus is encoded using apply nodes (@), lambda nodes (λ), sharing nodes ($\bigtriangledown$), weakening nodes ($\odot$), and croissants ($\frown$). A box allows a subgraph to be duplicated by a sharing node, or discarded by a weakening node. When sharing is no longer required, a croissant can open the box, allowing interaction with the subgraph inside. The meaning of the other nodes should be intuitive.

One port of each node or box is designated as the *principal port*. Other ports are *auxiliary ports*. Reduction takes place when two graph constructs are connected at their principal ports. A box can also interact with another box at its auxiliary port. Global reduction rules are shown in Figure 1, where black dots indicate principal ports. Graphs can also be reduced by local reduction rules, described elsewhere [1, 12]. Local reduction of the graph of a λ-term implements Lévy's *optimal reduction* [18].

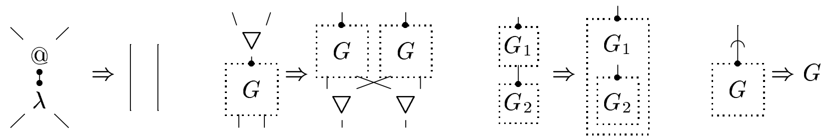


Fig. 1. Global reduction rules

The big question in implementing λ-calculus within this framework is where to put the boxes to allow the unrestricted sharing of values. We mention two commonly used boxing disciplines; see [19] for others. The *call-by-value (CBV) coding* boxes the graph of every λ-abstraction. Correspondingly, a croissant is placed on the function position of every apply node. The *call-by-name (CBN) coding* boxes the graph of the argument of every function application. Correspondingly, a variable reference is implemented by a croissant. These codings amount to Curry-Howard style embeddings of intuitionistic logic in linear logic. Figure 2 illustrates the CBN coding of the λ-calculus, which we will use in this paper. Note that the left side of a lambda node leads to the graph of the body, while the right side leads to the (perhaps shared) occurrence of the bound variable. Correspondingly, the left side of an apply node leads to the context of the application, while the right side leads to the argument. Our results are equally applicable to the CBV coding, and to any other consistent boxing strategy.

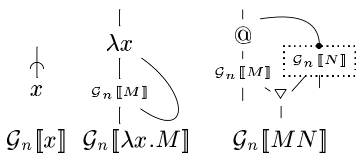


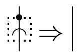
Fig. 2. CBN coding of λ-terms

Graphs of simply-typed λ-terms can be assigned linear-logic types. In particular, the type of a box is $!A$, allowing sharing of the A-typed value inside. Regardless of the boxing strategy, the constraints of linear logic typing imply:

Proposition 1. *Boxes never get in the way of β reductions.*

As a consequence, optimal (local) reduction reduces any two graphs with the same arrangement of apply, lambda, and sharing nodes in the same way.

Asperti has proposed some optimizations to these boxing strategies [1]. The simplest is to apply the following rule to the translation of a λ-term:



We apply this optimization, without comment, throughout the paper.

Implementing control operators: The above codings make the continuation and argument of an application equally accessible, the former on the left side, and the latter on the right side of the apply node. References to these values are similiarly equally accessible to a λ-abstraction, at the left and right side of the lambda node. Because the λ-calculus can only express the sharing of arguments, via parameter binding, the boxing strategies only ensure that the value of the argument is boxed. Control operators such as Scheme's `call/cc`, however, introduce the possibility to name, and thus duplicate and discard, the continuation.

3 Continuations in the λ-calculus

We now derive a family of graph encodings for terms in the λ-calculus with `call/cc` from encodings of the corresponding CPS terms. For pure λ-terms, this approach produces graphs with the same arrangement of lambda, apply, and sharing nodes as previous translations, and thus such graphs reduce to normal form in the optimal number of beta steps, as given by Lévy's specification.

3.1 The CPS transformation

Continuation-passing style (CPS) is a style of programming in which the continuation at each point is represented explicitly as a function. Because the continuation function makes explicit the remaining computation at the current program point, a CPS term necessarily specifies an evaluation order. A λ-calculus program can be converted to CPS automatically using a CPS transformation. Furthermore, the control operator `call/cc` can be translated into CPS. Typically a CPS transformation encodes a CBV or CBN evaluation order, however any consistent mixture is possible [15].

Plotkin's CBV and CBN CPS transformations, extended with the translation of `call/cc`, are shown in Figures 3 and 4, respectively [24]. For typed terms, these transformations induce a corresponding transformation on types. Define $\alpha^* \equiv \alpha$ for any base type α (including $\perp$) and $(\alpha \to \beta)^* \equiv \alpha^* \to \neg\neg\beta^*$ where $\neg\tau \equiv \tau \to \perp$; then the CBV CPS transformation maps a derivation $\{x : \sigma \in \mathit{\Gamma}\} \vdash E : \tau$ to $\{x : \sigma^* \mid x : \sigma \in \mathit{\Gamma}\} \vdash \mathcal{C}_v[\![E]\!] : \neg\neg\tau^*$. Similarly, define $\alpha^\dagger \equiv \alpha$, and $(\alpha \to \beta)^\dagger \equiv \neg\neg\alpha^\dagger \to \neg\neg\beta^\dagger$; then the CBN CPS transformation maps the same derivation to $\{x : \neg\neg\sigma^\dagger \mid x : \sigma \in \mathit{\Gamma}\} \vdash \mathcal{C}_n[\![E]\!] : \neg\neg\tau^\dagger$.

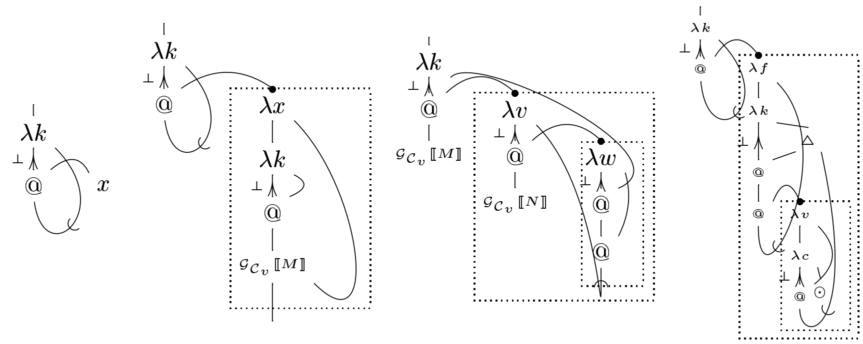
$$
\begin{aligned}
\mathcal{C}_v[\![x]\!] &\equiv \lambda\kappa.\kappa x \\
\mathcal{C}_v[\![\lambda x.M]\!] &\equiv \lambda\kappa.\kappa(\lambda x.\lambda k.\mathcal{C}_v[\![M]\!]k) \\
\mathcal{C}_v[\![MN]\!] &\equiv \lambda\kappa.\mathcal{C}_v[\![M]\!](\lambda v.\mathcal{C}_v[\![N]\!](\lambda w.vw\kappa)) \\
\mathcal{C}_v[\![\texttt{call/cc}]\!] &\equiv \lambda\kappa.\kappa(\lambda f.\lambda k.f(\lambda v.\lambda c.kv)k)
\end{aligned}
$$

Fig. 3. CBV CPS transformation of the λ-calculus, including `call/cc`

Replacing the λ-terms produced by a CPS transformation by the corresponding graphs gives a translation of terms into graphs in which the continuation is accessible as a sharable value. Figure 5 presents the graph translation corresponding to the CBV CPS transformations. We have used the CBN boxing strategy, although any strategy can be used. The translation corresponding to the CBN CPS transformation is similar. Only $\perp$ types are indicated.

$$
\begin{aligned}
\mathcal{C}_n[\![x]\!] &\equiv \lambda\kappa.x\kappa \\
\mathcal{C}_n[\![\lambda x.M]\!] &\equiv \lambda\kappa.\kappa(\lambda x.\lambda k.\mathcal{C}_n[\![M]\!]k) \\
\mathcal{C}_n[\![MN]\!] &\equiv \lambda\kappa.\mathcal{C}_n[\![M]\!](\lambda v.v(\mathcal{C}_n[\![N]\!])\kappa) \\
\mathcal{C}_n[\![\texttt{call/cc}]\!] &\equiv \lambda\kappa.\kappa(\lambda f.\lambda k.f(\lambda a.a(\lambda q.q(\lambda v.\lambda c.vk))k))
\end{aligned}
$$

Fig. 4. CBN CPS transformation of the λ-calculus, including `call/cc`



$\mathcal{G}_{\mathcal{C}_v}[\![x]\!]$ $\mathcal{G}_{\mathcal{C}_v}[\![\lambda x.M]\!]$ $\mathcal{G}_{\mathcal{C}_v}[\![MN]\!]$ $\mathcal{G}_{\mathcal{C}_v}[\![\texttt{call/cc}]\!]$

Fig. 5. Graph translation based on the CBV CPS transformation, and the CBN boxing strategy

This implementation strategy, while straightforward, is unsatisfactory. The CPS transformation introduces lambda and apply nodes that are not part of the original term. Thus, optimal reduction of the resulting graph does not reduce the original term using the minimal number of β steps. Indeed, the number of β steps is affected by the CPS transformation chosen. Furthermore, the graph translation does not exploit the symmetry between the left side of a lambda or apply node, which connects to the continuation of a function application, and the right side, which connects to the argument. The CPS encoding does, however, produce a graph in which continuations are consistently boxed. Thus, we would like a graph translation generating the same arrangement of lambda and apply nodes for pure λ-terms as the translations defined in Section 2, while retaining the boxing of continuations suggested by the CPS transformation.

3.2 The DS transformation on graphs

Essentially, we would like to eliminate the lambda and apply nodes that construct and manipulate continuations. To simplify the graph, we exploit the $\perp$ return type of every continuation and continuation abstraction. In a CPS program, we are not interested in the result of type $\perp$, but instead in the value passed to the initial continuation. We can show that the computation of a value of a non-$\perp$ type cannot depend on an edge transmitting a value of $\perp$ type. The simple conclusion is to remove all such edges. Because this transformation eliminates continuations from CPS graphs, we refer to it as the *direct-style (DS) transformation.*

Removing edges from the graph of course affects the nodes incident upon these edges. Figure 6 shows transformation rules sufficient to treat CPS terms. $\perp$-typed edges in other positions are treated similarly.

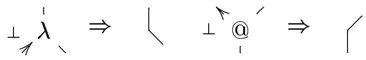


Fig. 6. DS graph transformation rules

While the graph codings of Section 2 were inspired by embeddings of intuitionistic logic into linear logic, they can equally well implement untyped terms. Here, however, we do require that $\perp$-typed values are used consistently.

The results of applying the DS transformation to the graph translations based on the CBV and CBN CPS transformations are shown in Figures 7 and 8, respectively. We refer to these translations as the $\mathrm{CBV}_{\mathrm{CPS/N}}$ and $\mathrm{CBN}_{\mathrm{CPS/N}}$ translations, respectively. Both achieve our goals: The arrangement of apply and lambda nodes in the translation of a pure λ-term is identical to that of the codings presented in Section 2, and continuations are boxed, allowing them to be duplicated or discarded.

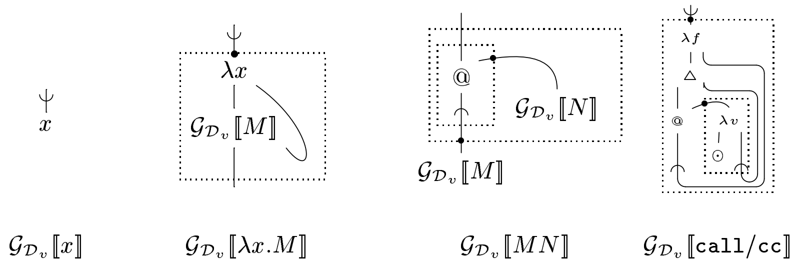
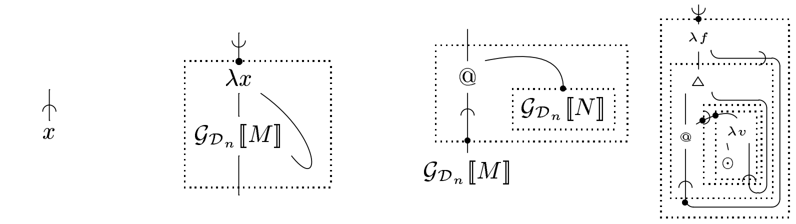


Fig. 7. DS graph translation derived from the CBV CPS transformation

3.3 Correctness of the DS transformation

A graph can be viewed as a set of apply and lambda nodes connected by edges that may contain sharing information, as controlled by sharing nodes, croissants, and box boundaries. Because the DS transformation only modifies (i.e.,



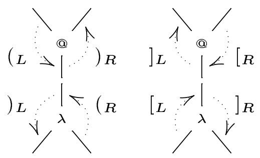
$\mathcal{G}_{\mathcal{D}_n}[\![x]\!]$ $\qquad$ $\mathcal{G}_{\mathcal{D}_n}[\![\lambda x.M]\!]$ $\qquad$ $\mathcal{G}_{\mathcal{D}_n}[\![MN]\!]$ $\qquad$ $\mathcal{G}_{\mathcal{D}_n}[\![\texttt{call/cc}]\!]$

Fig. 8. DS graph translation derived from the CBN CPS transformation

eliminates) apply and lambda nodes, it does not directly affect this sharing information. Consider the path between two edges that are ultimately connected by reduction. We prove that the DS transformation maintains the way in which such a path traverses apply and lambda nodes. Because the arrangement of the other nodes is not modified by the DS transformation, they continue to behave in the same way as well.

Throughout, we assume the graph is simply typed.

Definition 1. *A* path *is a directed sequence of connected edges, labelled as shown below, where for each node traversed on the path, the two edges incident on the node connect respectively to the principal port, and to an auxiliary port. The label of a path is the concatenation of the labels of the edges.*



Definition 2. *A* well-balanced path *is a path whose label is described by the following grammar, where* 1 *is the empty word:*

$$B ::= 1 \mid ({}_L B)_L \mid ({}_R B)_R \mid [{}_L B]_L \mid [{}_R B]_R \mid BB$$

An unbalanced path *is a path that is not well-balanced and whose label is a subword of a label derivable from B.*

This definition of well-balanced path generalizes that of Asperti and Laneve [2] to include paths that cannot occur in the translation of an ordinary λ-term, but can occur in the image of the DS transformation. Note that some paths are neither well-balanced nor unbalanced, for example a path that enters an apply node on the left and immediately exits the next lambda node on the right, with label $({}_L]_R$. Unbalanced paths describe correct information flow (in the sense of the geometry of interaction) in a well-typed graph, but cannot reduce to form a beta redex.

Proposition 2. *Let p be a path that is converted to an edge by the DS transformation. Then, p is either well-balanced or unbalanced.*

We can show the following by induction on the structure of a path in a well-typed graph:

Proposition 3. *In a well-balanced path, the first and last edges have identical types.*

An unbalanced path has a label of the form $B)X$ or $X(B$, where $)$ and $($ are one of the four forms of open/closed parentheses, and X is a well-balanced or unbalanced path. Applying Proposition 3 to B, we can show:

Proposition 4. *In an unbalanced path, either the first or last edge has arrow type.*

For a path p, if $\mathcal{D}(p)$ is an edge, every node of p must be eliminated by the DS transformation. Thus, using Proposition 4, we can show:

Proposition 5. *Let p be an unbalanced path. If $\mathcal{D}(p)$ is an edge, then either the first or last edge of p has type $A \to \perp$, for some A.*

Theorem 1. (Soundness) *Let G be the graph of a pure λ-term. Then the diagram*

$$\begin{tikzcd} G \arrow[r] \arrow[d, "\mathcal{D}"] & G' \arrow[d, "\mathcal{D}"] \\ \mathcal{D}(G) \arrow[r] & \mathcal{D}(G') \equiv G'' \end{tikzcd}$$

commutes, where:

1. *(The top path can be simulated by the bottom path): If G reduces to G' by global reductions, then $\mathcal{D}(G)$ reduces to $\mathcal{D}(G')$ by global reductions.*
2. *(The bottom path can be simulated by the top path): If $\mathcal{D}(G)$ reduces to G'' by global reductions, then there is some G' such that G reduces to G' by global reductions, and $G'' \equiv \mathcal{D}(G')$.*

Proof. To prove that the top path can be simulated by the bottom path, we use induction on the number of reduction steps in the top path. Observe that the effect of the DS transformation on a node is completely local, determined only by the types of edges incident on the node. Thus, the effect of the DS transformation on source nodes not affected by the reduction step is the same in both the source graph and the reduced graph; we need only consider the effect of the DS transformation on the nodes involved in the reduction step. Consider each possible redex in the source graph:

- A β-redex with function type $A \to B$, where $B \neq \perp$. Since the redex is not affected by the DS transformation, we can perform the same reduction step in $\mathcal{D}(G)$, and the resulting graph has the form $\mathcal{D}(G')$.

- A β-redex with function type $A \to \bot$. Reducing a beta redex creates an edge of the argument type (connecting the argument to the occurrence of the parameter) and an edge of the return type (connecting the result of the body to the context of the application). Thus, reducing a redex with function type $A \to \bot$ creates an edge of type A and an edge of type $\bot$ in G'. The edge of type $\bot$ is then eliminated by the DS transformation. Applying the DS transformation to G directly also eliminates the $\bot$-typed edges and connects the A-typed edges, thus having the same effect as beta reduction.
- Box duplication, absorption, croissant-box interaction. These operations each involve an edge of type $!A$, which is unaffected by the DS transformation. Thus, the identical operation can be performed in $\mathcal{D}(G)$.

To show that reductions in the bottom path can be simulated by reductions in the top path, we proceed by induction on the number of reduction steps from $\mathcal{D}(G)$ to G''. Since the DS transformation only converts λ- and apply nodes to edges, an edge e between two interaction ports in $\mathcal{D}(G)$ corresponds to a path p in G consisting of a sequence of lambda and apply nodes, such that $\mathcal{D}(p) = e$. If p is not an edge, we show that p must β-reduce to one; thus, a single reduction step in $\mathcal{D}(G)$ is simulated by a sequence of β-steps in G, followed by the same reduction step as performed in $\mathcal{D}(G)$. Because p consists of only λ- and apply nodes, it suffices to show that p is a balanced path. Consider each possible redex in $\mathcal{D}(G)$:

- A β-redex. If p were unbalanced, by Proposition 5, either the first or last edge would have type $A \to \bot$, for some A. In that case, the lambda or apply node connected to that edge would be eliminated by the DS transformation, contradicting the fact that p connects nodes that form a beta redex in $\mathcal{D}(G)$. Thus, p must be well-balanced.
- Box duplication, box absorption, croissant-box interaction. In all of these cases, the type of the edge between the interaction ports is $!A$. Because the boxes, croissants, and sharing nodes are not affected by the DS transformation, the first and last edges of p must also have type $!A$. By Proposition 4, p cannot be unbalanced. Thus, by Proposition 2, p must be well-balanced.

3.4 Embeddings of classical logic

Each of our proofnet implementations implicitly encodes, via an extended Curry-Howard correspondence, an embedding of classical logic in multiplicative-exponential linear logic (MELL). This family of encodings results from the mix-and-match of standard double-negation embeddings of classical logic into intuitionistic logic, composed with embeddings of intuitionistic logic into MELL. We discuss these constructions for minimal implicational logic with $\neg\neg$-elimination.

Let $[\alpha \to \beta] \equiv ![\alpha] \multimap [\beta]$ be the Girard translation of intuitionistic implication in linear logic; similarly, let $\langle \alpha \to \beta \rangle \equiv !(\langle \alpha \rangle \multimap \langle \beta \rangle)$ be the Gonthier-Abadi-Lévy translation (see [12]). By standard linear logic identities, $[\neg\neg\tau] = ?![\tau]$ and $\langle \neg\neg\tau \rangle = !?\langle \tau \rangle$, where ! and ? are the (dual) exponential modalities.

What does this mean in terms of graph reduction? When a subgraph G with root typed $?!\alpha$ is substituted into a sharable context C, the ? marks a croissant at the root of G that breaks the box around C, which then shares the value of type $!\alpha$. Dually, if G has type $!?\beta$, the context has type $?!(\beta)^{\perp}$, and the protocol for box opening and sharing reverses the role of context and value.

Recall $(\alpha \to \beta)^* \equiv \alpha^* \to \neg\neg\beta^*$ is the translation of $\alpha \to \beta$ induced by the CBV CPS transformation, and $(\alpha \to \beta)^\dagger \equiv \neg\neg\alpha^\dagger \to \neg\neg\beta^\dagger$ is the translation induced by the CBN CPS transformation. The CPS translations of a function (classical proof) $E : \alpha \to \beta$ result in a function of type $\neg\neg(\alpha \to \beta)^*$ or $\neg\neg(\alpha \to \beta)^\dagger$; we then have

$$
\begin{array}{ll}
[\neg\neg(\alpha \to \beta)^*] = ?!(!\,[\alpha^*] \multimap ?!\,[\beta^*]) & \langle\neg\neg(\alpha \to \beta)^*\rangle = !?!(\langle\alpha^*\rangle \multimap !?\langle\beta^*\rangle) \\
\left[\neg\neg(\alpha \to \beta)^\dagger\right] = ?!(!?!\left[\alpha^\dagger\right] \multimap ?!\left[\beta^\dagger\right]) & \langle\neg\neg(\alpha \to \beta)^\dagger\rangle = !?!(!?\langle\alpha^\dagger\rangle \multimap !?\langle\beta^\dagger\rangle)
\end{array}
$$

Other variants of this mix-and-match style are possible. Fewer modalities mean fewer boxes and greater implementation efficiency.

4 Related work

We consider three areas of related work: other CPS transformations, other approaches to converting CPS programs back to direct style, and other connections between control operators and linear logic.

Optimizing the CPS transformation: The Plotkin CPS transformations create many "administrative" redexes involving the application of a continuation or continuation abstraction [24]. Our DS transformation converts administrative redexes into box-croissant redexes, adding a bureaucratic cost to optimal reduction [1, 20]. More optimized CPS transformations [7, 25] could generate more efficient implementations.

Converting CPS programs back to direct style: Danvy first investigated the problem of converting a CPS program back to direct style [5], later extended with Lawall. The conversions were only on terms that could be output by CPS transformation. Our DS transformation also relies on a uniform, but weaker property: values of type $\perp$ must occur consistently, and do not contribute to the final result. At the extreme, our DS transformation is simply the identity on the graphs of DS terms.

Relating languages with control operators to linear logic: Nishizaki also investigated encodings of λ-calculus plus `call/cc` in proofnets [22]. He showed that normalization of these proofnets is complete with respect to normalization in the term language. He began by adding modalities in an ad hoc manner (induced mechanically by our $\mathrm{CBV}_{\mathrm{CPS/N}}$ translation) to the type $!A \multimap B$, allowing sharing of both values and continuations. His more complex translation is an optimization of our $\mathrm{CBV}_{\mathrm{CPS/N}}$ translation, eliminating some box croissant interactions corresponding to administrative redexes. Because Nishizaki derived a translation from the types rather than from the semantics, he had to prove that the resulting graphs model the semantics of the language. The correctness of our approach relies only on the correctness of the CPS transformation, and on the correctness of the DS transformation on graphs, which is independent of the language being implemented.

In a sequel to his earlier work on symmetric λ-calculus, Filinski used linear logic as a tool for understanding continuations [9]. Some of the linear types he proposed for continuations appear in our codings, the most common being $?!\alpha$, resulting from the DS transformation of a graph with type $(\alpha \multimap \perp) \multimap \perp$. Griffin, and later Murthy, showed the relation between so-called $\neg\neg$-embeddings of classical logic in intuitionistic logic, and the implementation of control operators [14, 21]. In particular, they showed how varieties of the CPS transformation provide the constructive content of such embeddings. We further translate such terms into proofnets in direct style, eliminating the administrative redexes. The result is a family of constructive embeddings of classical logic into linear logic.

5 Future work

Since the continuation created by `call/cc` is abortive, a term containing `call/cc` can reduce to different normal forms; its CPS counterpart, like all pure λ-terms, has only one normal form. Because the DS transformation produces boxes according to the CPS transformation providing its input, it should be possible to identify, among the shared normal forms it can return, the answer that would have been produced by the CPS-converted input. We leave further analyses of these observations to future work.

Efficiently managing the reified continuation is a significant problem in implementing languages with control operators [4, 16]. Proofnet implementations suggest the possibility of evaluating programs containing control operators using optimal reduction, with a minimal copying of shared values. Nevertheless, we are exchanging the savings of optimal reduction for the overhead of box management. Further experiments are needed to understand whether the exchange is cost-effective, and if it can be further optimized by better box technology.

Our proofnet technology might be extended to languages with functional control operators, such as Danvy and Filinski's `shift` and `reset`, and Sitaram and Felleisen's `control` and `prompt` [6, 27]. While `shift` and `reset` are defined in terms of a CPS transformation, continuations do not have return type $\perp$; the DS transformation is then inapplicable. Both `shift` and `reset`, and `control` and `prompt` can be defined in terms of `call/cc` and a reference cell [10, 27]. Bawden has shown how to implement reference cells using sharing graphs [3], so this strategy may still lead to an interesting proofnet implementation.

6 Conclusions

We have shown how to implement various languages with explicit control using graph reduction, where the structure of the graphs are proofnets from linear logic. The principal technical difficulty in such codings is the location of boxes, which allow computations to be shared. Rather than specifying a fixed scheme for locating boxes, we have introduced a general methodology based on the CPS transform. Different versions of CPS, followed by our DS transform on graphs, produce a wide range of consistent schemes for locating boxes in proofnets. The noble art of linear decorating, to repeat the phrase of Schellinx [26], has been replaced by a factory.

The theoretical foundation of our implementation technology means that we have a consistent semantics provided by the geometry of interaction, and a means of incrementally evaluating continuations via optimal evaluation. The codings may clarify full abstraction theorems for languages with explicit control, given the full completeness results that are known for linear logic. But the genuine progress reflected in the presented techniques is the technology transfer of logic and proofnets to the mundane algorithmics of implementation. The pragmatics of double negation in logic, for example, is just *packaging:* the boxing of sharable data so that they can interact with each other. Further implementation improvements amount to a better understanding of where to put boxes. A generation of compiler writers has spent considerable effort optimizing the efficiency of sharing expressions. We have presented a systematic basis on which to optimize the sharing of continuations, providing new territory for similar efficiency improvements.

Acknowledgments. We thank Alan Bawden and Olivier Danvy for commenting on a draft of this paper.

References

1. A. Asperti. $\delta\circ!\epsilon = 1$: Optimizing optimal λ-calculus implementations. In *Rewriting Techniques and Applications*, pages 102–116, Kaiserslautern, Germany, 1995.
2. A. Asperti and C. Laneve. Paths, computations and labels in the lambda-calculus. In *Rewriting Techniques and Applications, 5th International Conference*, volume 690, pages 152–167. Lecture Notes in Computer Science, 1993.
3. A. Bawden. Implementing distributed systems using linear naming. TR 1627, MIT AI Lab, March 1993. (PhD thesis, originally titled *Linear Graph Reduction: Confronting the Cost of Naming*. MIT, May 1992).
4. C. Bruggeman, O. Waddell, and R.K. Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, volume 31(5), pages 99–107, Philadephia, Pennsylvania, May 1996. SIGPLAN Notices.
5. O. Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.
6. O. Danvy and A. Filinski. Abstracting control. In *Proceeding of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.
7. O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 4:361–391, 1992.
8. A. Filinski. Declarative continuations and categorical duality. Technical Report 89/11, University of Copenhagen, 1989. Masters Thesis.
9. A. Filinski. Linear continuations. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 27–38, Albuquerque, New Mexico, January 1992.

10. A. Filinski. Representing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994.
11. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 46:1–102, 1986.
12. 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*, pages 15–26, Albuquerque, New Mexico, January 1992.
13. G. Gonthier, M. Abadi, and J.-J. Lévy. Linear logic without boxes. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science*, pages 223–234, June 1992.
14. T. Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990.
15. J. Hatcliff and O. Danvy. A generic account of continuation-passing styles. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994.
16. R. Hieb, R.K. Dybvig, and C. Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, volume 25(6), pages 66–77, White Plains, New York, June 1990. SIGPLAN Notices.
17. J. Lamping. An algorithm for optimal lambda calculus reduction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16–30, San Francisco, California, January 1990.
18. J.-J. Lévy. *Optimal Reductions in the Lambda-Calculus*, pages 159–191. Academic Press, 1980.
19. I. Mackie. *The Geometry of Implementation*. PhD thesis, University of London, September 1994.
20. I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 117–128, Baltimore, Maryland, September 1998.
21. C.R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Cornell University, August 1990.
22. S. Nishizaki. Programs with continuations and linear logic. *Science of Computer Programming*, 21(2):165–190, 1993.
23. M. Parigot. Lambda-mu-calculus: an algorithmic interpretation of classical natural deduction. In A. Voronkov, editor, *Logic Programming and Automated Reasoning: International Conference LPAR '92 Proceedings, St. Petersburg, Russia*, pages 190–201, Berlin, DE, 1992. Springer-Verlag.
24. G. D. Plotkin. Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
25. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, November 1993.
26. H. Schellinx. *The Noble Art of Linear Decorating*. PhD thesis, Institute for Logic, Language and Computation, University of Amsterdam, 1994.
27. D. Sitaram and M. Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.