An optimised term graph rewriting engine for $\mathcal{X}$

Measuring the cost of α-conversion

Steffen van Bakel and Jayshan Raghunandan

Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ, UK
{svb,jr200}@doc.ic.ac.uk

Abstract

This paper studies the calculus $\mathcal{X}$, that has its foundation in Classical Logic; we present an implementation for $\mathcal{X}$ using term graph rewriting techniques, and discuss improvements thereof which result in an increasingly more efficient running of the reduction engine. We show that name capture can be dealt with 'on the fly', by realising the avoidance of capture through adding or modifying the rewrite rules. We study two different approaches to garbage collection, and compare the various implementations by presenting benchmarks.

1 Introduction

This paper presents a term graph rewriting implementation for the (untyped) calculus $\mathcal{X}$ as first presented in [4], and studies the efficiency of a number of approaches towards implementation issues, in particular that of α-conversion. $\mathcal{X}$ is a sequent calculus which embodies both substitution and context call; it was first been defined in [24, 25, 18] and was later extensively studied in [2, 3].

The origin of $\mathcal{X}$ lies within the quest for a Curry-Howard correspondence to Gentzen's sequent calculus LK for Classical Logic, introduced in [14]. In particular, $\mathcal{X}$ corresponds to the implicative fragment of Kleene's G_3 [17], with implicit weakening and contraction. LK is a logical system in which the rules only introduce connectives (but on either side of a sequent), in contrast to *natural deduction* (also introduced in [14]) which uses rules that introduce or eliminate connectives in the logical formulae. Natural deduction normally derives statements with a single conclusion, whereas LK allows for multiple conclusions, deriving sequents of the form $A_1,\ldots,A_n \vdash B_1,\ldots,B_m$, where $A_1,\ldots,A_n$ is to be understood as $A_1\wedge\ldots\wedge A_n$ and $B_1,\ldots,B_m$ is to be understood as $B_1\vee\ldots\vee B_m$. G_3 has four rules: *axiom*, *left introduction* of the arrow, *right introduction*, and *cut*.

$$(Ax):\ \frac{}{\Gamma, A \vdash A, \Delta} \qquad (cut):\ \frac{\Gamma \vdash A, \Delta \quad \Gamma, A \vdash \Delta}{\Gamma \vdash \Delta}$$

$$(\Rightarrow R):\ \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta} \qquad (\Rightarrow L):\ \frac{\Gamma \vdash A, \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \Rightarrow B \vdash \Delta}$$

Since LK has only *introduction* rules (or better, no *elimination* rules), the only way to eliminate a connective is to eliminate the whole formula in which it appears via an application of the (cut)-rule. Gentzen defined an informal procedure that eliminates all applications of the (cut)-rule from a proof of a sequent, generating a proof in *normal form* of the same sequent, that is, without a cut; this procedure is defined via local reductions of the proof tree which essentially correspond to innermost reduction.

Starting from different approaches in that area, in [18] Lengrand introduced $\mathcal{X}$ as a calculus that enjoys the Curry-Howard isomorphism for G_3; similar work was presented by Urban [24, 25], who defined an expressive restriction of cut-elimination that is strongly normalisable.

$\mathcal{X}$ can be seen as an extension of Urban's calculus, and was studied in [2, 3] in the context of a number of calculi. A notable difference with Urban's work is that he set out to study the structure of proofs, so the terms of his calculus carry types and correspond only to proofs. In the approach to $\mathcal{X}$ of [2, 3] which we take also here, we study terms *without* types, and drop the condition that terms should represent proofs of the logic altogether: we study a pure calculus. This opens the research to notions as recursion, normalising strategies, confluence, etc.

The calculus $\mathcal{X}$ achieves a Curry-Howard isomorphism for the proofs in LK by constructing *witnesses* for derivable sequents. In establishing the isomorphism, similar to calculi like Parigot's $\lambda\mu$ [20] and $\overline{\lambda}\mu\tilde{\mu}$ [13], Roman names are attached to formulae in the left context, and Greek names for those on the right, and syntactic structure is associated to the rules. Names on the left can be seen as inputs to the term, and names to the right as outputs; since multiple formulae can appear on both sides, this implies that a term can not only have more than one input, but also more than one output. There are two kinds of names (connectors) in $\mathcal{X}$: *sockets* (inputs, with Roman names, that are reminiscent of values) and *plugs* (outputs, with Greek names, that are reminiscent of continuations), that correspond to *variables* and *co-variables*, respectively, in [28], or, alternatively, to Parigot's λ and μ-variables [20].

Gentzen's proof reductions by cut-elimination become the fundamental principle of computation in $\mathcal{X}$. Cuts in proofs are witnessed by $P\widehat{\alpha} \dagger \widehat{x}Q$ (called the *cut* of P and Q via α and x), and the reduction rules specify how to remove them: a term is in normal form if and only if it has no sub-term of this shape. The generalisation with respect to Gentzen's proof reduction is that now all cuts can be eliminated, not just those that are innermost.

$\mathcal{X}$ as an untyped language provides an excellent general purpose machine, very well suited to encode various calculi; [3] illustrates the expressive power of $\mathcal{X}$ by giving consistent interpretations of the Lambda Calculus [12, 5], Bloo and Rose's calculus of explicit substitution $\lambda\mathbf{x}$ [10], $\lambda\mu$, and $\overline{\lambda}\mu\tilde{\mu}$.

Perhaps the main feature of $\mathcal{X}$ is that it constitutes a *variable*, *application*, and *substitution*-free method of computation. Rather than having variables like x representing places where terms can be inserted, in $\mathcal{X}$ the symbol x represents a *socket*, to which a term can be *attached* via a *plug* α; both plugs and sockets carry *names*, and the only substitution-like operation is that of *renaming*. The definition of reduction on $\mathcal{X}$ constitutes of rules that describe the reorganisation of a term in small steps, and shows the subtle interaction between plugs and sockets.

Reduction in $\mathcal{X}$ can be specified as a *conditional term rewriting system*; the only non-standard aspects is the presence of binding structures. It is this observation that led us to the research we report on in this paper; i.e. the building of an interpreter for $\mathcal{X}$ using the term graph rewriting technology. A tool was developed that allows users to input terms from $\mathcal{X}$, reducing cuts at will in $\mathcal{X}$'s full reduction system, or to limit reduction to either call-by-name or call-by-value.

We not only set out to study $\mathcal{X}$ and its properties, but also focus on trying to extend $\mathcal{X}$ into a true programming language. With that in mind, we have concentrated on building an efficient interpreter, and sought different, increasingly better solutions to garbage collection, capture avoidance and name capture. In [4] we reported on the first results of our implementation efforts. In particular, to avoid problems caused by nested binding of connectors, a lazy copy mechanism was introduced, and almost all rewrite rules were defined using this.

In this paper we will improve on this result, and show that the special character of $\mathcal{X}$, being a conditional term rewriting system based on renaming, makes it possible to study α-conversion *on the level of the language itself*. So in $\mathcal{X}$, α-conversion is not just a hidden implementation issue, but a first-class citizen. This allows us to directly measure the cost of α-conversion using the same currency as for the cost of cut-elimination: with minor changes

to the rules, we can test our different solutions on the level of the language itself.

This paper does not claim to find innovative solutions to the problem of capture itself; many solutions to this problem exists, and we have, so far, chosen standard approaches. What we do achieve is a the creation of a platform on which it is easy *to compare* these various solutions in terms of execution cost, thus enabling justification for the choice of a certain technology over another.

We will present improvements on the reduction engine in the tool as first presented in [4], and show that these are expressible directly in $\mathcal{X}$ itself. We will present three solutions for the problem of capture that are both expressed as changes to the rules: the first will use the rebinding of bound connectors, the second will preserve Barendregt's convention, whereas the third avoids the capture of free connectors by binders. Much to our surprise, the resulting rewrite rules of the latter two solutions are very similar, but for the fact that in the last freeness is an issue, rather than being bound as in the second. The reduction engine thus obtained proved to be impressively much more efficient though, especially after the addition of two different notions of garbage collection.

Outline of this paper. This paper is organised as follows. In Section 2 we will repeat the $\mathcal{X}$-calculus, and some of its main properties. In Section 3, we will show that more expressive rules regarding garbage collection and deactivation can be safely added to the calculus. Section 4 presents the term graph rewriting engine that is used for our implementation, and in Section 5 we discuss three ways to deal with α-conversion. In Section 6 we formally define our reduction strategy combinators, that help us implement our three solutions in comparable ways, and give our measurement results.

2 The $\mathcal{X}$-calculus

In this section we will give the definition of the $\mathcal{X}$-calculus [18] that was proven to be a fine-grained implementation model for various well-known calculi in [2]. $\mathcal{X}$ features two separate categories of 'connectors', *plugs* and *sockets*, that act as input and output channels.

As mentioned in the introduction, $\mathcal{X}$ enjoys the Curry-Howard relation with Kleene's variant G_3 of Gentzen's LK. This is achieved by labelling the formulae with term information, and building witnesses for proofs by associating derivation rules to syntactical constructs. In the construction of the witness of a logical statement, when in applying a rule a premise or conclusion disappears from the sequent, the corresponding name gets bound in the term that is constructed, and when a premise or conclusion gets created, a different, free (often new) name is associated to it. For example, in the creation of the term for right introduction of the arrow

$$\frac{P :\cdot \Gamma, x{:}A \vdash_{\mathcal{X}} \alpha{:}B, \Delta}{\widehat{x} P \widehat{\alpha} \cdot \beta :\cdot \Gamma \vdash_{\mathcal{X}} \beta{:}A \to B, \Delta}$$

the input x and the output α are bound, and β is free. This case is interesting in that it highlights a special feature of $\mathcal{X}$, not found in other calculi. In (applicative) calculi related to natural deduction, like the λ-calculus, only inputs are named, and the linking to a term that will be inserted is done via λ-abstraction and application. The output (i.e. result) on the other hand is anonymous; where a term 'moves to' carries a name via a *variable* that acts as a pointer to the positions where the term is to be inserted, but where it comes from is not mentioned, since it is implicit. Since in $\mathcal{X}$ a term P can have many inputs and outputs, it is unsound to consider P a function; however, fixing *one* input x and *one* output α, we can see P as a function 'from x to α'. We make this limited view of P available via the output β, thereby *exporting* 'P is a function from x to α'. The types given to the connectors confirm this view, which is also

supported by the logic.

Definition 2.1 (SYNTAX) The terms of the $\mathcal{X}$-calculus are defined by the following grammar, where x, y range over the infinite set of *sockets*, and α, β over *plugs*.

$$P, Q ::= \underset{\textit{capsule}}{\langle x \cdot \alpha \rangle} \mid \underset{\textit{export}}{\widehat{y} P \widehat{\beta} \cdot \alpha} \mid \underset{\textit{import}}{P \widehat{\beta} [y] \widehat{x} Q} \mid \underset{\textit{cut}.}{P \widehat{\alpha} \dagger \widehat{x} Q}$$

Notice that $\mathcal{X}$ has no notion of abstraction; in particular, $P\widehat{\beta}$ and $\widehat{x}Q$ are not terms.

The $\widehat{\cdot}$ symbolises that the socket (plug) underneath is bound in the term directly to the right (left). The unconnected inputs and outputs in a term make up the collection of *free connectors*, that are *inactive* during a computational step.

Definition 2.2 (FREE AND BOUND) The sets of *free* and *bound sockets* and *free* and *bound plugs* in a term are defined by:

$$\begin{array}{lll|lll}
fs(\langle x \cdot \alpha \rangle) & = & \{x\} & bs(\langle x \cdot \alpha \rangle) & = & \emptyset \\
fs(\widehat{y} P \widehat{\beta} \cdot \alpha) & = & fs(P) \backslash \{y\} & bs(\widehat{y} P \widehat{\beta} \cdot \alpha) & = & bs(P) \cup \{y\} \\
fs(P \widehat{\beta} [y] \widehat{x} Q) & = & fs(P) \cup (fs(Q) \backslash \{x\}) & bs(P \widehat{\beta} [y] \widehat{x} Q) & = & bs(P) \cup bs(Q) \cup \{x\} \\
fs(P \widehat{\alpha} \dagger \widehat{x} Q) & = & fs(P) \cup (fs(Q) \backslash \{x\}) & bs(P \widehat{\alpha} \dagger \widehat{x} Q) & = & bs(P) \cup bs(Q) \cup \{x\} \\
\\
fp(\langle x \cdot \alpha \rangle) & = & \{\alpha\} & bp(\langle x \cdot \alpha \rangle) & = & \emptyset \\
fp(\widehat{x} P \widehat{\alpha} \cdot \beta) & = & fp(P) \backslash \{\alpha\} & bp(\widehat{x} P \widehat{\alpha} \cdot \beta) & = & bp(P) \cup \{\alpha\} \\
fp(P \widehat{\alpha} [y] \widehat{x} Q) & = & (fp(P) \backslash \{\alpha\}) \cup fp(Q) & bp(P \widehat{\alpha} [y] \widehat{x} Q) & = & bp(P) \cup \{\alpha\} \cup bp(Q) \\
fp(P \widehat{\alpha} \dagger \widehat{x} Q) & = & (fp(P) \backslash \{\alpha\}) \cup fp(Q) & bp(P \widehat{\alpha} \dagger \widehat{x} Q) & = & bp(P) \cup \{\alpha\} \cup bp(Q)
\end{array}$$

The set of *bound connectors* is defined as $bc(P) = bs(P) \cup bp(P)$, and we sometimes write, for example, $bs(P, Q)$ as shorthand for $bs(P) \cup bs(Q)$; we use the notation $fc(P)$ $(= fs(P) \cup fp(P))$ for the free connectors.

Note that, in $\widehat{x}\langle x \cdot \alpha \rangle \widehat{\alpha} \cdot \alpha$, α is bound *and* free. The standard way to avoid this anomaly is to assume Barendregt's convention, which states that *free* and *bound* names should be distinct. As illustrated in Section 5, this then becomes an implementation issue, which we will address here for $\mathcal{X}$. As usual, we will identify terms that only differ in the names of bound connectors (modulo α-conversion, as usual). To maintain this convention during reduction, implicit α-conversion needs to take place, which is the main issue we tackle in this paper (see Section 5).

2.1 Reduction on $\mathcal{X}$

The calculus, defined by the reduction rules below, explains in detail how cuts are propagated through terms to be eventually evaluated at the level of capsules.

The intuition behind reduction is: the cut $P \widehat{\alpha} \dagger \widehat{x} Q$ expresses the intention to connect all αs in P and xs in Q, and reduction will realise this by either connecting all αs to all xs (if x does not exist in Q, P will disappear), or all xs to all αs (if α does not exist in P, Q will disappear).

Reduction is defined by specifying both the interaction between well-connected syntactic structures, and how to deal with propagating active nodes to points in the term where they can interact. This strongly depends on the following notion.

Definition 2.3 (INTRODUCTION OF CONNECTORS) *P introduces x* : Either $P = Q \widehat{\beta} [x] \widehat{y} R$ and $x \notin fs(Q, R)$, or $P = \langle x \cdot \alpha \rangle$.
P introduces α : Either $P = \widehat{x} Q \widehat{\beta} \cdot \alpha$ and $\alpha \notin fp(Q)$, or $P = \langle x \cdot \alpha \rangle$.

We consider P and Q in $P\widehat{\alpha} \dagger \widehat{x}Q$ well connected when both α and x are introduced.

The main reduction rules are:

Definition 2.4 (LOGICAL RULES) The logical rules are presented by (where both α and x are introduced in the cuts):

$$\begin{array}{rrcl}
(cap): & \langle y{\cdot}\alpha\rangle\widehat{\alpha} \dagger \widehat{x}\langle x{\cdot}\beta\rangle & \to & \langle y{\cdot}\beta\rangle \\
(exp): & (\widehat{y}P\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \dagger \widehat{x}\langle x{\cdot}\gamma\rangle & \to & \widehat{y}P\widehat{\beta}{\cdot}\gamma \\
(imp): & \langle y{\cdot}\alpha\rangle\widehat{\alpha} \dagger \widehat{x}(P\widehat{\beta}[x]\widehat{z}Q) & \to & P\widehat{\beta}[y]\widehat{z}Q \\
(exp\text{-}imp): & (\widehat{y}P\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\gamma}[x]\widehat{z}R) & \to & \begin{cases} Q\widehat{\gamma} \dagger \widehat{y}(P\widehat{\beta} \dagger \widehat{z}R) \\ (Q\widehat{\gamma} \dagger \widehat{y}P)\widehat{\beta} \dagger \widehat{z}R \end{cases}
\end{array}$$

The first three logical rules above specify a renaming (reconnecting) procedure, whereas the last rule specifies the basic computational step: it links the export of a function, available on the plug α, to an adjacent import via the socket x.

In case the connectors in a cut are *not* introduced, the logical rules cannot be directly applied, and the propagation rules specified below come into play. Applying these will either try to connect the socket to the plug, or the plug to the socket. This choice is expressed via the *activation* of a cut, represented by the tilting of a dagger.

Definition 2.5 (ACTIVE CUTS) The syntax is extended with two *flagged* or *active* cuts:

$$P, Q \;::=\; \ldots \mid P\widehat{\alpha} \nearrow \widehat{x}Q \mid P\widehat{\alpha} \nwarrow \widehat{x}Q$$

Terms constructed without these flagged cuts are called *pure*.

A right-activated cut $P\widehat{\alpha} \nwarrow \widehat{x}Q$ corresponds to the substitution of P for x via α, and will attempt to connect all αs to all xs, whereas the left-activated cut $P\widehat{\alpha} \nearrow \widehat{x}Q$ is its dual, the substitution of Q for α via x, and will attempt to connect all xs to all αs.

Definition 2.6 (ACTIVATING) We define two cut-activation rules.

$$\begin{array}{rrcll}
(a\nearrow): & P\widehat{\alpha} \dagger \widehat{x}Q & \to & P\widehat{\alpha} \nearrow \widehat{x}Q & \text{if } P \text{ does not introduce } \alpha \\
(\nwarrow a): & P\widehat{\alpha} \dagger \widehat{x}Q & \to & P\widehat{\alpha} \nwarrow \widehat{x}Q & \text{if } Q \text{ does not introduce } x
\end{array}$$

An activated cut is processed by 'pushing' it systematically through the syntactic structure of the term in the direction indicated by the tilting of the dagger, as described by the propagation rules below. The pushing of the active cut continues until the level of capsules is reached, where it is either deactivated or destroyed.

Definition 2.7 (PROPAGATION RULES) **Left propagation**:

$$\begin{array}{rrcll}
(d\nearrow): & \langle y{\cdot}\alpha\rangle\widehat{\alpha} \nearrow \widehat{x}P & \to & \langle y{\cdot}\alpha\rangle\widehat{\alpha} \dagger \widehat{x}P & \\
(cap\nearrow): & \langle y{\cdot}\beta\rangle\widehat{\alpha} \nearrow \widehat{x}P & \to & \langle y{\cdot}\beta\rangle, & (\beta \neq \alpha) \\
(exp\text{-}outs\nearrow): & (\widehat{y}Q\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \nearrow \widehat{x}P & \to & (\widehat{y}(Q\widehat{\alpha} \nearrow \widehat{x}P)\widehat{\beta}{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{x}P, & (\gamma \textit{ fresh}) \\
(exp\text{-}ins\nearrow): & (\widehat{y}Q\widehat{\beta}{\cdot}\gamma)\widehat{\alpha} \nearrow \widehat{x}P & \to & \widehat{y}(Q\widehat{\alpha} \nearrow \widehat{x}P)\widehat{\beta}{\cdot}\gamma, & (\gamma \neq \alpha) \\
(imp\nearrow): & (Q\widehat{\beta}[z]\widehat{y}R)\widehat{\alpha} \nearrow \widehat{x}P & \to & (Q\widehat{\alpha} \nearrow \widehat{x}P)\widehat{\beta}\,[z]\,\widehat{y}(R\widehat{\alpha} \nearrow \widehat{x}P) & \\
(cut\nearrow): & (Q\widehat{\beta} \dagger \widehat{y}R)\widehat{\alpha} \nearrow \widehat{x}P & \to & (Q\widehat{\alpha} \nearrow \widehat{x}P)\widehat{\beta} \dagger \widehat{y}(R\widehat{\alpha} \nearrow \widehat{x}P) &
\end{array}$$

Right propagation:

$$\begin{array}{rll}
(\nwarrow d): & P\widehat{\alpha}\nwarrow\widehat{x}\langle x\cdot\beta\rangle & \to\ P\widehat{\alpha}\dagger\widehat{x}\langle x\cdot\beta\rangle \\
(\nwarrow cap): & P\widehat{\alpha}\nwarrow\widehat{x}\langle y\cdot\beta\rangle & \to\ \langle y\cdot\beta\rangle, \qquad (y\neq x) \\
(\nwarrow exp): & P\widehat{\alpha}\nwarrow\widehat{x}(\widehat{y}Q\widehat{\beta}\cdot\gamma) & \to\ \widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\beta}\cdot\gamma \\
(\nwarrow imp\text{-}outs): & P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}[x]\widehat{y}R) & \to P\widehat{\alpha}\dagger\widehat{z}((P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R)), \\
 & & \qquad (z\ fresh) \\
(\nwarrow imp\text{-}ins): & P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}[z]\widehat{y}R) & \to\ (P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R), \quad (z\neq x) \\
(\nwarrow cut): & P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \to\ (P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\beta}\dagger\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R)
\end{array}$$

Whenever an active cut meets a term exhibiting the connector it is trying to communicate with, a new (inactive) cut, with that connector made fresh, is 'created', as in rules $(exp\text{-}outs\nearrow)$ and $(\nwarrow imp\text{-}outs)$, representing an attempt to communicate at this level. This new inactive cut can reduce via a logical rule, or pushing can continue in the other direction; notice that the freshly created connector is introduced in $(exp\text{-}outs\nearrow)$ and $(\nwarrow imp\text{-}outs)$.

Right-propagation is reminiscent of *substitution* of terms for term-variables; left-propagation $P\widehat{\alpha}\nearrow\widehat{x}Q$ then is its dual: it expresses the connection of the continuation Q, accessible via x, to all the 'calls' α in P.

Definition 2.8 We write

$\to$: for the reduction relation generated by the contextual closure of the logical, propagation and activation rules;

$\to^*$: for the reflexive, transitive closure of $\to$;

$P \overset{\mathcal{X}}{=} Q$: if P and Q have exactly the same normal forms[1].

In [2] some basic properties were shown, which essentially show that the calculus is well behaved.

Lemma 2.9 (GARBAGE COLLECTION AND RENAMING [2])

$$\begin{array}{llll}
P\widehat{\alpha}\nearrow\widehat{x}Q\ \to^*\ P & (\alpha\notin fp(P), P\ pure) & P\widehat{\delta}\dagger\widehat{z}\langle z\cdot\alpha\rangle\ \to^*\ P[\alpha/\delta] & (P\ pure) \\
Q\widehat{\alpha}\nwarrow\widehat{x}P\ \to^*\ P & (x\notin fs(P), P\ pure) & \langle z\cdot\alpha\rangle\widehat{\alpha}\dagger\widehat{x}P\ \to^*\ P[z/x] & (P\ pure)
\end{array}$$

In Lemma 3.5 and 3.6, below, we will show that we can drop the restriction that P has to be pure.

The reduction relation $\to$ is not confluent; this comes in part from the critical pair that activates a cut $P\widehat{\alpha}\dagger\widehat{x}Q$ of pure P and Q in two ways: if P does not contain (so does not introduce) α and Q does not contain x; then we have both

$$\begin{array}{lll}
P\widehat{\alpha}\dagger\widehat{x}Q\ \to\ (a\nearrow) & & P\widehat{\alpha}\dagger\widehat{x}Q\ \to\ (\nwarrow a) \\
P\widehat{\alpha}\nearrow\widehat{x}Q\ \to^*\ (2.9) & \text{and} & P\widehat{\alpha}\nwarrow\widehat{x}Q\ \to^*\ (2.9) \\
P & & Q
\end{array}$$

In [2] two sub-systems of reduction are defined, that correspond to call-by-name (CBN) and call-by-value (CBV) reduction.

Definition 2.10 *i*) *Call by value* reduction $P\to_{\mathrm{V}}Q$ is defined as the reduction system obtained by replacing rules $(\nwarrow a)$ and $(exp\text{-}imp)$ by:

$$\begin{array}{rll}
(\nwarrow a_{\mathrm{V}}): & P\widehat{\alpha}\dagger\widehat{x}Q\ \to\ P\widehat{\alpha}\nwarrow\widehat{x}Q, & (\textit{if } P \textit{ introduces } \alpha \textit{ and} \\
 & & \quad Q \textit{ does not introduce } x) \\
(exp\text{-}imp_{\mathrm{V}}): & (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}R)\ \to\ Q\widehat{\gamma}\dagger\widehat{y}(P\widehat{\beta}\dagger\widehat{z}R) &
\end{array}$$

[1] Since reduction is $\mathcal{X}$ is non-confluent, normal forms are not unique.

ii) Likewise, we define *call by name* reduction $P \to_{\mathrm{N}} Q$ as the reduction system obtained by replacing rules $(a\not\leftarrow)$ and $(\mathit{exp\text{-}imp})$ by:

$$(a\not\leftarrow_{\mathrm{N}}): \qquad P\widehat{\alpha} \dagger \widehat{x}Q \;\to\; P\widehat{\alpha} \not\leftarrow \widehat{x}Q, \;\; (\textit{if } Q \textit{ introduces } x \textit{ and } P \textit{ does not introduce } \alpha)$$

$$(\mathit{exp\text{-}imp}_{\mathrm{N}}): \; (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\gamma}[x]\widehat{z}R) \;\to\; (Q\widehat{\gamma} \dagger \widehat{y}P)\widehat{\beta} \dagger \widehat{z}R$$

If a cut can be activated in two ways, the CBV strategy only allows to activate it via $(a\not\leftarrow)$; the CBN strategy can only activate such a cut via $(\not\rightarrow a)$.

This way, we obtain two notions of reduction that are clearly confluent: all rules are left-linear and non-overlapping.

2.2 Typing for $\mathcal{X}$

The notion of type assignment on $\mathcal{X}$ corresponds to the basic implicative system G_3 for Classical Logic.

Definition 2.11 (TYPES AND CONTEXTS [2]) *i*) The set of types[2], ranged over by A, B, is defined over a set of *type-variables* ranged over by φ, by the grammar:

$$A, B \;::=\; \varphi \mid A \to B$$

ii) A *context of sockets* Γ is a mapping from sockets to types, denoted as a finite set of *statements* $x{:}A$, such that the *subjects* of the statements (the sockets) are distinct. We write $\Gamma, x{:}A$ for the context $\Gamma \cup \{x{:}A\}$.

Contexts of plugs Δ and the notation $\alpha{:}A, \Delta$ are defined similarly.

Definition 2.12 (TYPING FOR $\mathcal{X}$ [2]) *i*) *Type judgements* are expressed via the ternary relation $P \vdots \Gamma \vdash \Delta$, where Γ is a context of sockets and Δ is a context of plugs, and P is a term. We say that P is the *witness* of this judgement.

ii) *Type assignment for* $\mathcal{X}$ is defined by the following sequent rules:

$$(ax): \frac{}{\langle x\cdot\alpha\rangle \vdots \Gamma, x{:}A \vdash \alpha{:}A, \Delta} \qquad (cut): \frac{P \vdots \Gamma \vdash \alpha{:}A, \Delta \quad Q \vdots \Gamma, x{:}A \vdash \Delta}{P\widehat{\alpha} \dagger \widehat{x}Q \vdots \Gamma \vdash \Delta}$$

$$(\Rightarrow R): \frac{P \vdots \Gamma, x{:}A \vdash \alpha{:}B, \Delta}{\widehat{x}P\widehat{\alpha}\cdot\beta \vdots \Gamma \vdash \beta{:}A \to B, \Delta} \qquad (\Rightarrow L): \frac{P \vdots \Gamma \vdash \alpha{:}A, \Delta \quad Q \vdots \Gamma, x{:}B \vdash \Delta}{P\widehat{\alpha}[y]\widehat{x}Q \vdots \Gamma, y{:}A \to B \vdash \Delta}$$

We write $P \vdots \Gamma \vdash \Delta$ if there exists a derivation built out of these rules that has this judgement in the bottom line.

Notice that, by the special meaning we associate to the comma in contexts, contraction is implicit. Also, since Γ and Δ are non-specific, so is weakening.

In $P \vdots \Gamma \vdash \Delta$, the term P acts as a *witness* of the judgement; Γ and Δ carry the types of the free connectors in P, as unordered sets.

The following result was shown in [2]:

Theorem 2.13 (WITNESS REDUCTION [2]) *If* $P \vdots \Gamma \vdash \Delta$, *and* $P \to Q$, *then* $Q \vdots \Gamma \vdash \Delta$.

[2]The types considered in this paper are normally known as *Curry* types.

2.3 $\mathcal{X}$ as a reduction machine for the λ-calculus

In [2] the relation between $\mathcal{X}$ and many other calculi is studied; as an illustration, in this section, we will briefly highlight the relation between the $\lambda\mathbf{x}$ and $\mathcal{X}$. We assume the reader to be familiar with the λ-calculus [5].

Definition 2.14 ($\lambda\mathbf{x}$ [10]) The set of $\lambda\mathbf{x}$-terms is defined by the grammar:

$$M, N \ ::= \ x \mid \lambda x.M \mid MN \mid M\langle x := N\rangle$$

A term of the form $M\langle x := N\rangle$ is called a *closure*. A term which contains no closure is called a *pure term*.

The set of *free* and *bound* variables in a term M, $fv(M)$ and $bv(M)$, are defined by:

$$\begin{array}{llll} fv(x) & = \{x\} & bv(x) & = \emptyset \\ fv(\lambda x.M) & = fv(M) \setminus \{x\} & bv(\lambda x.M) & = bv(M) \cup \{x\} \\ fv(MN) & = fv(M) \cup fv(N) & bv(MN) & = bv(M) \cup bv(N) \\ fv(M\langle x := N\rangle) & = fv(M) \setminus \{x\} \cup fv(N) & bv(M\langle x := N\rangle) & = bv(M) \cup bv(N) \cup \{x\} \end{array}$$

We accept Barendregt's convention on free and bound variables, so assume that no variable appears both free and bound in a term.

Definition 2.15 (REDUCTION IN $\lambda\mathbf{x}$ [10]) Reduction in $\lambda\mathbf{x}$ is defined through the rules:

$$\begin{array}{rrcll} (\mathsf{B}): & (\lambda x.M)P & \to & M\langle x := P\rangle & \\ (\mathsf{VarI}): & x\langle x := P\rangle & \to & P & \\ (\mathsf{VarK}): & y\langle x := P\rangle & \to & y, & (y \neq x) \\ (\mathsf{App}): & (MN)\langle x := P\rangle & \to & M\langle x := P\rangle\, N\langle x := P\rangle & \\ (\mathsf{Abs}): & (\lambda y.M)\langle x := P\rangle & \to & \lambda y.M\langle x := P\rangle & \end{array}$$

The λx_{gc} calculus is defined by adding a rule for garbage collection. We point out that this rule requires the use of a more complicated side-condition.

$$(\mathsf{gc}) \quad M\langle x := P\rangle \ \to \ M, \quad (x \notin fv(M))$$

In fact, the rules (VarK) and (gc) overlap.

To illustrate the expressive power of $\mathcal{X}$, we now define the direct encoding of $\lambda\mathbf{x}$-terms into $\mathcal{X}$, inspired by Gentzen's encoding of natural deduction into the sequent calculus [14].

Definition 2.16 (INTERPRETING THE λ-CALCULUS [2]) The interpretation of λ-terms into terms of $\mathcal{X}$ in the context α, $\mathbb{[}M\mathbb{]}^{\mathrm{G}}_{\alpha}$, is defined by:

$$\begin{array}{rcll} \mathbb{[}x\mathbb{]}^{\mathrm{G}}_{\alpha} & = & \langle x \cdot \alpha\rangle & \\ \mathbb{[}\lambda x.M\mathbb{]}^{\mathrm{G}}_{\alpha} & = & \widehat{x}\,\mathbb{[}M\mathbb{]}^{\mathrm{G}}_{\beta}\widehat{\beta}\cdot\alpha, & (\beta \textit{ fresh}) \\ \mathbb{[}MN\mathbb{]}^{\mathrm{G}}_{\alpha} & = & \mathbb{[}M\mathbb{]}^{\mathrm{G}}_{\gamma}\widehat{\gamma} \dagger \widehat{x}(\mathbb{[}N\mathbb{]}^{\mathrm{G}}_{\beta}\widehat{\beta}\ [x]\ \widehat{y}\langle y \cdot \alpha\rangle), & (x, y, \beta, \gamma \textit{ fresh}) \\ \mathbb{[}M\langle x := N\rangle\mathbb{]}^{\mathrm{G}}_{\alpha} & = & \mathbb{[}N\mathbb{]}^{\mathrm{G}}_{\gamma}\widehat{\gamma} \nwarrow \widehat{x}\,\mathbb{[}M\mathbb{]}^{\mathrm{G}}_{\alpha}, & (\gamma \textit{ fresh}) \end{array}$$

For this encoding, [2] shows:

Theorem 2.17 ([2]) *If* $\Gamma \vdash_{\lambda} M : A$, *then* $\mathbb{[}M\mathbb{]}^{\mathrm{G}}_{\alpha} :\cdot\ \Gamma \vdash \alpha : A$.

In [2], also the following relation is shown between reduction in $\lambda\mathbf{x}$ and $\mathcal{X}$:

Theorem 2.18 ([2]) *i) If* $M \to_{\mathbf{x}} N$, *then* $\mathbb{[}M\mathbb{]}^{\mathrm{G}}_{\alpha} \to^{*} \mathbb{[}N\mathbb{]}^{\mathrm{G}}_{\alpha}$.

ii) If $M \to_\beta N$, *then* $\llbracket M \rrbracket_\alpha^{\mathrm{G}} \to^* \llbracket N \rrbracket_\alpha^{\mathrm{G}}$.

Alternatively, we can consider Prawitz' encoding of natural deduction into classical logic [21, 23]. This encoding preserves normal forms when interpreting a particular test case (i.e. if a λ-term is in normal form, then the interpretation in the $\mathcal{X}$-calculus is also in normal form), and will also allow us to indirectly obtain a cost of using the $\mathcal{X}$-calculus to reduce arbitrary λ-terms in Section 6.

Definition 2.19 (PRAWITZ' INTERPRETATION) There are three parts to the interpretation. We use the symbol ':' to represent the list concatenation operator and '[ ]' for the empty list. In addition, we use the symbols L to represent lists of λ-terms, and the symbols M, N, P to represent arbitrary λ-terms.

$$
\begin{array}{rcl}
\llbracket x \rrbracket_\alpha^{\mathrm{P}} & = & \langle x \cdot \alpha \rangle \\
\llbracket \lambda x.M \rrbracket_\alpha^{\mathrm{P}} & = & \widehat{x} \llbracket M \rrbracket_\beta^{\mathrm{P}} \widehat{\beta} \cdot \alpha \\
\llbracket MN \rrbracket_\alpha^{\mathrm{P}} & = & \llbracket MN, [\,] \rrbracket_\alpha^{\mathrm{P}} \\
\llbracket M\langle x := N \rangle \rrbracket_\alpha^{\mathrm{P}} & = & \llbracket N \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{x} \llbracket N \rrbracket_\alpha^{\mathrm{P}} \\
\llbracket xN, L \rrbracket_\alpha^{\mathrm{P}} & = & \llbracket N \rrbracket_\beta^{\mathrm{P}} \widehat{\beta}\ [x]\ \widehat{y} \llbracket L \rrbracket_\alpha^{\mathrm{P}} y \\
\llbracket (\lambda x.M)N, L \rrbracket_\alpha^{\mathrm{P}} & = & \llbracket \lambda x.M \rrbracket_\beta^{\mathrm{P}} \widehat{\beta} \dagger \widehat{y} (\llbracket N \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma}\ [y]\ \widehat{z} \llbracket L \rrbracket_\alpha^{\mathrm{P}} z) \\
\llbracket MNP, L \rrbracket_\alpha^{\mathrm{P}} & = & \llbracket MN, P : L \rrbracket_\alpha^{\mathrm{P}} \\
\llbracket [\,] \rrbracket_\alpha^{\mathrm{P}} x & = & \langle x \cdot \alpha \rangle \\
\llbracket M : L \rrbracket_\alpha^{\mathrm{P}} x & = & \llbracket M \rrbracket_\beta^{\mathrm{P}} \widehat{\beta}\ [x]\ \widehat{y} \llbracket L \rrbracket_\alpha^{\mathrm{P}} y
\end{array}
$$

For this encoding, we can show the expected results.

Theorem 2.20 *i) If* $M \to_{\mathbf{x}} N$, *then* $\llbracket M \rrbracket_\alpha^{\mathrm{P}} \to^* \llbracket N \rrbracket_\alpha^{\mathrm{P}}$.
ii) If $M \to_\beta^* N$, *then* $\llbracket M \rrbracket_\alpha^{\mathrm{P}} \to^* \llbracket N \rrbracket_\alpha^{\mathrm{P}}$.

Proof: By induction on the definition of one-step explicit reduction, of which we show the more interesting cases.

$(\lambda x.M)N \to_\beta M\langle x := N \rangle$: $\llbracket (\lambda x.M)N \rrbracket_\alpha^{\mathrm{P}} \triangleq$
$\llbracket (\lambda x.M)N, [\,] \rrbracket_\alpha^{\mathrm{P}} \triangleq$
$\llbracket \lambda x.M \rrbracket_\beta^{\mathrm{P}} \widehat{\beta} \dagger \widehat{y} (\llbracket N \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma}\ [y]\ \widehat{z} \llbracket [\,] \rrbracket_\alpha^{\mathrm{P}} z) \triangleq$
$(\widehat{x} \llbracket M \rrbracket_\delta^{\mathrm{P}} \widehat{\delta} \cdot \beta) \widehat{\beta} \dagger \widehat{y} (\llbracket N \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma}\ [y]\ \widehat{z} \langle z \cdot \alpha \rangle) \to (\mathit{exp\text{-}imp})$
$\llbracket N \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \dagger \widehat{x} (\llbracket M \rrbracket_\delta^{\mathrm{P}} \widehat{\delta} \dagger \widehat{z} \langle z \cdot \alpha \rangle) \to (2.9)$
$\llbracket N \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \dagger \widehat{x} \llbracket M \rrbracket_\alpha^{\mathrm{P}} \to (\nwarrow a)$
$\llbracket N \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{x} \llbracket M \rrbracket_\alpha^{\mathrm{P}} \triangleq \llbracket M\langle x := N \rangle \rrbracket_\alpha^{\mathrm{P}}$

$x\langle x := P \rangle \to P$: $\llbracket x\langle x := P \rangle \rrbracket_\alpha^{\mathrm{P}} \triangleq \llbracket P \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{x} \langle x \cdot \alpha \rangle \to^* (\nwarrow d, 2.9)\ \llbracket P \rrbracket_\alpha^{\mathrm{P}}$

$y\langle x := P \rangle \to y, \mathit{if}\ y \neq x$: $\llbracket y\langle x := P \rangle \rrbracket_\alpha^{\mathrm{P}} \triangleq \llbracket P \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{x} \langle y \cdot \alpha \rangle \to^* (2.9)\ \langle y \cdot \alpha \rangle \triangleq \llbracket y \rrbracket_\alpha^{\mathrm{P}}$

$(MN)\langle x := P \rangle \to M\langle x := P \rangle N\langle x := P \rangle$: $\llbracket (MN)\langle x := P \rangle \rrbracket_\alpha^{\mathrm{P}} \triangleq$
$\llbracket P \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{x} \llbracket MN \rrbracket_\alpha^{\mathrm{P}} \triangleq \llbracket P \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{x} \llbracket MN, [] \rrbracket_\alpha^{\mathrm{P}}$
We now distinguish the cases:
$M = x$: $\llbracket P \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{x} \llbracket xN, [] \rrbracket_\alpha^{\mathrm{P}} \triangleq$
$\llbracket P \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{x} (\llbracket N \rrbracket_\beta^{\mathrm{P}} \widehat{\beta}\ [x]\ \widehat{z} \langle z \cdot \alpha \rangle) \to (\nwarrow \mathit{imp\text{-}outs}), (2.9)$
$\llbracket P \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{v} ((\llbracket P \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{x} \llbracket N \rrbracket_\beta^{\mathrm{P}}) \widehat{\beta}\ [v]\ \widehat{z} \langle z \cdot \alpha \rangle) \triangleq$
$\llbracket P \rrbracket_\gamma^{\mathrm{P}} \widehat{\gamma} \nwarrow \widehat{v} \llbracket vN\langle x := P, [\,] \rangle \rrbracket_\alpha^{\mathrm{P}} \triangleq$
$\llbracket vN\langle x := P \rangle \langle v := P \rangle, [\,] \rrbracket_\alpha^{\mathrm{P}} \triangleq \llbracket xN\langle x := P \rangle, [\,] \rrbracket_\alpha^{\mathrm{P}}$

$$
\begin{array}{lll}
M = y, y \neq x: & \llbracket P \rrbracket^{\mathrm{P}}_{\gamma} \widehat{\gamma} \nwarrow \widehat{x} \llbracket y N, [\,] \rrbracket^{\mathrm{P}}_{\alpha} & \triangleq \\
& \llbracket P \rrbracket^{\mathrm{P}}_{\gamma} \widehat{\gamma} \nwarrow \widehat{x} (\llbracket N \rrbracket^{\mathrm{P}}_{\beta} \widehat{\beta}\, [y]\, \widehat{z} \langle z \cdot \alpha \rangle) & \to \ (\nwarrow \mathit{imp\text{-}ins}), (2.9) \\
& (\llbracket P \rrbracket^{\mathrm{P}}_{\gamma} \widehat{\gamma} \nwarrow \widehat{x} \llbracket N \rrbracket^{\mathrm{P}}_{\beta}) \widehat{\beta}\, [y]\, \widehat{z} \langle z \cdot \alpha \rangle & \triangleq \\
& \llbracket N \langle x := P \rangle \rrbracket^{\mathrm{P}}_{\beta} \widehat{\beta}\, [y]\, \widehat{z} \langle z \cdot \alpha \rangle & \triangleq \ \llbracket y N \langle x := P \rangle, [\,] \rrbracket^{\mathrm{P}}_{\alpha}
\end{array}
$$

The other cases follow by induction.

The second part follows from the fact that $\to_{\mathbf{x}}$ implements $\to_{\beta}$. □

3 Optimising reduction

We should point out that, using the rules above, not all typeable terms are strongly normalisable. For example, to allow for the propagation of cuts over cuts immediately leads to non-termination, since we can always choose the outermost cut as the one to contract. Although the notion of cut-elimination as proposed here has no rule that would allow this behaviour, it can be mimicked, which can lead to non-termination for typeable terms, as already observed by Urban [24].

Take $P \widehat{\alpha} \dagger \widehat{x} (\langle x \cdot \beta \rangle \widehat{\beta} \dagger \widehat{z} Q)$, such that $x \notin fs(Q), \beta \notin fp(P)$, and P, Q pure, then:

$$
\begin{array}{ll}
P \widehat{\alpha} \dagger \widehat{x} (\langle x \cdot \beta \rangle \widehat{\beta} \dagger \widehat{z} Q) & \to^* \ (\nwarrow a), (\nwarrow \mathit{cut}) \\
(P \widehat{\alpha} \nwarrow \widehat{x} \langle x \cdot \beta \rangle) \widehat{\beta} \dagger \widehat{z} (P \widehat{\alpha} \nwarrow \widehat{x} Q) & \to^* \ (\nwarrow d), (2.9) \\
(P \widehat{\alpha} \dagger \widehat{x} \langle x \cdot \beta \rangle) \widehat{\beta} \dagger \widehat{z} Q & \to^* \ (a \nearrow), (\mathit{cut} \nearrow) \\
(P \widehat{\beta} \nearrow \widehat{z} Q) \widehat{\alpha} \dagger \widehat{x} (\langle x \cdot \beta \rangle \widehat{\beta} \nearrow \widehat{z} Q) & \to^* \ (d \nearrow), (2.9) \\
P \widehat{\alpha} \dagger \widehat{x} (\langle x \cdot \beta \rangle \widehat{\beta} \dagger \widehat{z} Q) &
\end{array}
$$

(example communicated by Alexander J. Summers)

Urban gives a solution for this unwanted reduction behaviour, and shows it sufficient to obtain strong-normalisation of typeable terms. He adds the rules

$$
\begin{array}{lll}
(P \widehat{\alpha} \dagger \widehat{x} \langle x \cdot \beta \rangle) \widehat{\beta} \nearrow \widehat{y} Q & \to & (P \widehat{\beta} \nearrow \widehat{y} Q) \widehat{\alpha} \dagger \widehat{y} Q \\
P \widehat{\alpha} \nwarrow \widehat{x} (\langle x \cdot \beta \rangle \widehat{\beta} \dagger \widehat{y} Q) & \to & P \widehat{\alpha} \dagger \widehat{y} (P \widehat{\alpha} \nwarrow \widehat{x} Q)
\end{array}
$$

and gives them priority over the rules $(\mathit{cut} \nearrow)$ and $(\nwarrow \mathit{cut})$ by changing those to

$$
\begin{array}{lll}
(P \widehat{\alpha} \dagger \widehat{x} Q) \widehat{\beta} \nearrow \widehat{y} R & \to & (P \widehat{\beta} \nearrow \widehat{y} R) \widehat{\alpha} \dagger \widehat{x} (Q \widehat{\beta} \nearrow \widehat{u} R), \quad Q \neq \langle x \cdot \beta \rangle \\
P \widehat{\alpha} \nwarrow \widehat{x} (Q \widehat{\beta} \dagger \widehat{y} R) & \to & (P \widehat{\alpha} \nwarrow \widehat{x} Q) \widehat{\beta} \dagger \widehat{y} (P \widehat{\alpha} \nwarrow \widehat{x} R), \quad Q \neq \langle x \cdot \beta \rangle
\end{array}
$$

Notice that the side-condition $Q \neq \langle x \cdot \beta \rangle$ is quite different in character from the rules for $\mathcal{X}$ we presented above, in that now *syntactic equality* between terms is tested, rather than just a syntactic property of a term.

We can remedy this by avoiding the deactivation of cuts altogether.

Definition 3.1 We can remove the rules $(d \nearrow)$ and $(\nwarrow d)$, and add the following rules:

$$
\begin{array}{lrcll}
(\mathit{flip} \nwarrow): & \langle z \cdot \alpha \rangle \widehat{\alpha} \nearrow \widehat{x} P & \to & \langle z \cdot \alpha \rangle \widehat{\alpha} \nwarrow \widehat{x} P, & (P \text{ does not introduce } x) \\
(\nearrow \mathit{imp}): & \langle y \cdot \alpha \rangle \widehat{\alpha} \nearrow \widehat{x} (P \widehat{\delta} [x] \widehat{z} Q) & \to & P \widehat{\delta} [y] \widehat{z} Q, & (x \text{ introduced}) \\
(\nearrow \mathit{cap}): & \langle z \cdot \alpha \rangle \widehat{\alpha} \nearrow \widehat{x} \langle x \cdot \beta \rangle & \to & \langle z \cdot \beta \rangle &
\end{array}
$$

$$
\begin{array}{lrcll}
(\nearrow \mathit{flip}): & P \widehat{\alpha} \nwarrow \widehat{x} \langle x \cdot \beta \rangle & \to & P \widehat{\alpha} \nearrow \widehat{x} \langle x \cdot \beta \rangle, & (P \text{ does not introduce } \alpha) \\
(\mathit{exp} \nwarrow): & (\widehat{y} P \widehat{\delta} \cdot \alpha) \widehat{\alpha} \nwarrow \widehat{x} \langle x \cdot \gamma \rangle & \to & \widehat{y} P \widehat{\delta} \cdot \gamma, & (\alpha \text{ introduced}) \\
(\mathit{cap} \nwarrow): & \langle x \cdot \alpha \rangle \widehat{\alpha} \nwarrow \widehat{y} \langle y \cdot \beta \rangle & \to & \langle x \cdot \beta \rangle &
\end{array}
$$

We do not need to check if a term matches another, nor need to give priority to rules.

This new set of rules does not change the end result of reduction:

Theorem 3.2 *Let $\rightarrow_o$ denote the notion of reduction obtained by the changes as suggested above, and assume P and Q are pure.*

i) If $P \rightarrow_o^* Q$, then $P \rightarrow_{\mathcal{X}}^* Q$.

ii) If $P \rightarrow_{\mathcal{X}}^* Q$ and Q is in normal form, then $P \rightarrow_o^* Q$.

Proof: Easy. □

In Section 6, we will present an evaluation strategy that will choose to run the inactive cut that is created by the deactivation rule immediately after its creation, thus side-stepping the problem.

The set of reduction rules can be optimised further than just those expressed in Lemma *2.9*. For example, the applicability of the garbage collection rules is limited, since they both involve pure terms. However, we are able to generalise these results to include terms with active cuts.

We aim to add more generic garbage collection rules; in fact, we will show their admissibility below (Theorem 3.5), for which we first need to show a number of results.

Lemma 3.3 *i) For all P, Q pure, there exists an R pure such that $P\widehat{\alpha} \not\prec \widehat{x}Q \rightarrow^* R$.*

ii) For all P, Q pure, there exists an R pure such that $P\widehat{\alpha} \not\setminus \widehat{x}Q \rightarrow^* R$.

Proof: *i*) By induction on structure of terms. We highlight one case:

$$\begin{array}{lll} P = \widehat{y}P'\widehat{\beta}\cdot\alpha: & (\widehat{y}P'\widehat{\beta}\cdot\alpha)\widehat{\alpha} \not\prec \widehat{x}Q & \rightarrow \\ & (\widehat{y}(P'\widehat{\alpha} \not\prec \widehat{x}Q)\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}Q & \rightarrow^* \quad (IH, R \textit{ pure}) \\ & (\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}Q & \end{array}$$

Notice that this last term is pure.

ii) By induction on structure of terms. We highlight again one case:

$$\begin{array}{lll} Q = Q_1\widehat{\beta}\,[x]\,\widehat{y}Q_2: & P\widehat{\alpha} \not\setminus \widehat{x}(Q_1\widehat{\beta}\,[x]\,\widehat{y}Q_2) & \rightarrow \quad (Q_1, Q_2 \textit{ pure}) \\ & P\widehat{\alpha} \dagger \widehat{v}((P\widehat{\alpha} \not\setminus \widehat{x}Q_1)\widehat{\beta}\,[v]\,\widehat{y}(P\widehat{\alpha} \not\setminus \widehat{x}Q_2)) & \rightarrow^* \quad (IH, R_1, R_2 \textit{ pure}) \\ & P\widehat{\alpha} \dagger \widehat{v}(R_1\widehat{\beta}\,[v]\,\widehat{y}R_2) & \end{array}$$

Notice again that this last term is pure. □

We can now use this lemma to give a stronger result.

Lemma 3.4 *For all $\mathcal{X}$-terms P, there exists a reduction path $P \rightarrow^* P'$, with P' pure.*

Proof: For each active cut in a term, we define its *depth* d as the distance (calculated in nodes) from the root of the tree that represents the term. For any particular depth of the term, we define its *group size*, g, as the number of active cuts at that depth of the tree. We define the *class*, c, of a term as the pair of the depth of the innermost cut and its group size: $c = \langle d, g \rangle$. We finish the proof by lexicographic induction on the class of a term.

If P is pure, then $P \equiv P'$. Otherwise: take the set (of size g) of innermost active cuts at depth d. There are two cases to consider:

$g > 1$: take any term T in this set, $T = R\widehat{\alpha} \not\prec \widehat{x}Q$ (or $T = R\widehat{\alpha} \not\setminus \widehat{x}Q$). Then R and Q are pure (being involved in an innermost flagged cut). By Lemma 3.3, we know $T \rightarrow^* S$ (pure). This eliminates the active cut from the proof, so the group-size (g) of the term reduces by one, and the class decreases.

$g = 1$: the active cut is eliminated from the term by Lemma 3.3. Since there are no more

active cuts at this level, the depth of the innermost cut decreases (to the next lowest), and the class reduces. □

We are now ready to justify some more general garbage collection rules, that essentially equates the nets $P\widehat{\alpha}\nearrow\widehat{x}Q$ and P, provided $\alpha\notin fp(P)$.

Lemma 3.5 (GENERALISED GARBAGE COLLECTION)
i) *If* $\alpha\notin fp(P)$, *then* $P\widehat{\alpha}\nearrow\widehat{x}Q \stackrel{\mathcal{X}}{=} P$.
ii) *If* $x\notin fs(Q)$, *then* $P\widehat{\alpha}\nwarrow\widehat{x}Q \stackrel{\mathcal{X}}{=} Q$.

Proof: *i*) For the second part ($nf(P)\subseteq nf(P\widehat{\alpha}\nearrow\widehat{x}Q)$), take $T\in nf\,P$, reduce $P\widehat{\alpha}\nearrow\widehat{x}Q$ to $T\widehat{\alpha}\nearrow\widehat{x}Q$, remark that T is pure, and apply Lemma *2.9*.

For the first, we show that if $P\widehat{\alpha}\nearrow\widehat{x}Q\rightarrow^{*}T$, then $P\rightarrow^{*}T$, where T is a normal form. We achieve this by showing that we can run a reduction on P, mimicking a reduction taking place on $P\widehat{\alpha}\nearrow\widehat{x}Q$ by essentially ignoring all reductions inside Q; the only problem might be when the presence of $[\,]\widehat{\alpha}\nearrow\widehat{x}Q$ disturbs the reduction behaviour.

The proof completes by co-induction (we only show some interesting cases):

• $P=(P_1\widehat{\beta}\,[v]\,\widehat{y}P_2)\widehat{\gamma}\dagger\widehat{z}P_3$. We can run $((P_1\widehat{\beta}\,[v]\,\widehat{y}P_2)\widehat{\gamma}\dagger\widehat{z}P_3)\widehat{\alpha}\nearrow\widehat{x}Q$ in a number of ways. Any reduction inside P_1,P_2 or P_3 is dealt with by induction, so we can focus on the cuts involved. Assume we apply rule $(cut\nearrow)$ to propagate the outermost cut and obtain

$$((P_1\widehat{\beta}\,[v]\,\widehat{y}P_2)\widehat{\alpha}\nearrow\widehat{x}Q)\widehat{\gamma}\dagger\widehat{z}(P_3\widehat{\alpha}\nearrow\widehat{x}Q)$$

Now the top-most (inactive) cut can be activated in two directions; let's go left:

$$((P_1\widehat{\beta}\,[v]\,\widehat{y}P_2)\widehat{\alpha}\nearrow\widehat{x}Q)\widehat{\gamma}\nearrow\widehat{z}(P_3\widehat{\alpha}\nearrow\widehat{x}Q)$$

This (outermost) activated cut cannot propagate, since the cut directly underneath it is active; propagating that first gives

$$((P_1\widehat{\alpha}\nearrow\widehat{x}Q)\widehat{\beta}\,[v]\,\widehat{y}(P_2\widehat{\alpha}\nearrow\widehat{x}Q))\widehat{\gamma}\nearrow\widehat{z}(P_3\widehat{\alpha}\nearrow\widehat{x}Q)$$

Now the top-cut can propagate, to give

$$((P_1\widehat{\alpha}\nearrow\widehat{x}Q)\widehat{\gamma}\nearrow\widehat{z}(P_3\widehat{\alpha}\nearrow\widehat{x}Q))\widehat{\beta}\,[v]\,\widehat{y}((P_2\widehat{\alpha}\nearrow\widehat{x}Q)\widehat{\gamma}\nearrow\widehat{z}(P_3\widehat{\alpha}\nearrow\widehat{x}Q))$$

By induction we can mimic $P_i\widehat{\alpha}\nearrow\widehat{x}Q$ by P_i, for $i\in\{1,2,3\}$. We can simulate this particular reduction on P as follows:

$$(P_1\widehat{\beta}\,[v]\,\widehat{y}P_2)\widehat{\gamma}\dagger\widehat{z}P_3\ \rightarrow^{*}(a\nearrow),(imp\nearrow)\ (P_1\widehat{\gamma}\nearrow\widehat{z}P_3)\widehat{\beta}\,[v]\,\widehat{y}(P_2\widehat{\gamma}\nearrow\widehat{z}P_3)$$

• $P=(\widehat{y}P_1\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{z}P_2$. As above, a reduction inside P_1 or P_2 creates no problems, so we can focus on the cuts involved. When we propagate the top cut, we get

$$((\widehat{y}P_1\widehat{\beta}\cdot\gamma)\widehat{\alpha}\nearrow\widehat{x}Q)\widehat{\gamma}\dagger\widehat{z}(P_2\widehat{\alpha}\nearrow\widehat{x}Q)$$

If we now left-activate the top cut, similar to above, we can only propagate the innermost cut, and obtain:

$$(\widehat{y}(P_1\widehat{\alpha}\nearrow\widehat{x}Q)\widehat{\beta}\cdot\gamma)\widehat{\gamma}\nearrow\widehat{z}(P_2\widehat{\alpha}\nearrow\widehat{x}Q)$$

Now applying rule $(exp\text{-}outs\nearrow)$ will give:

$$(\widehat{y}((P_1\widehat{\alpha}\nearrow\widehat{x}Q)\widehat{\gamma}\nearrow\widehat{z}(P_2\widehat{\alpha}\nearrow\widehat{x}Q))\widehat{\beta}\cdot\delta)\widehat{\delta}\dagger\widehat{z}(P_2\widehat{\alpha}\nearrow\widehat{x}Q)$$

Now, if γ is introduced in $\widehat{y}P_1\widehat{\beta}\cdot\gamma$, then γ does not appear free inside P_1 nor in Q, and, by induction, we can assume that $(P_1\widehat{\alpha}\not\leftarrow\widehat{x}Q)\widehat{\gamma}\not\leftarrow\widehat{z}(P_2\widehat{\alpha}\not\leftarrow\widehat{x}Q)$ can be simulated by P_1, and that $P_2\widehat{\alpha}\not\leftarrow\widehat{x}Q$ can be simulated by P_2. We can mimic this reduction with a reduction path of length zero, performing an α-conversion:

$$(\widehat{y}P_1\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{z}P_2 \to (\widehat{y}P_1\widehat{\beta}\cdot\delta)\widehat{\delta}\dagger\widehat{z}P_2$$

If γ is not introduced, we still can assume that $P_i\widehat{\alpha}\not\leftarrow\widehat{x}Q$ can be simulated by P_i, for $i\in\{1,2\}$, and we can simulate this reduction on P via a series of steps:

$$(\widehat{y}P_1\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{z}P_2 \to (\widehat{y}P_1\widehat{\beta}\cdot\gamma)\widehat{\gamma}\not\leftarrow\widehat{z}P_2 \to (\widehat{y}(P_1\widehat{\gamma}\not\leftarrow\widehat{z}P_2)\widehat{\beta}\cdot\delta)\widehat{\delta}\dagger\widehat{z}P_2$$

- All other cases are shown in a similar fashion.

So every reduction to a normal form, starting from $P\widehat{\alpha}\not\leftarrow\widehat{x}Q$, can be mimicked by reducing P, so every normal form of $P\widehat{\alpha}\not\leftarrow\widehat{x}Q$, can be reached from P.

ii) Similar. □

This result now helps to justify more general deactivation rules.

Theorem 3.6 (GENERALISED DEACTIVATION) *i*) $P\widehat{\alpha}\not\leftarrow\widehat{x}Q \overset{\mathcal{X}}{=} P\widehat{\alpha}\dagger\widehat{x}Q$ *, if* P *introduces* α*.*
ii) $P\widehat{\alpha}\not\rightarrow\widehat{x}Q \overset{\mathcal{X}}{=} P\widehat{\alpha}\dagger\widehat{x}Q$ *, if* Q *introduces* x*.*

Proof: *i*) If P introduces α, we have two cases:
$P=\langle x\cdot\alpha\rangle$: By rule $(d\not\leftarrow)$.
$P=\widehat{x}P'\widehat{\beta}\cdot\alpha$: Then $\alpha\notin fp(P)$, and

$$\begin{array}{lll}(\widehat{x}P'\widehat{\beta}\cdot\alpha)\widehat{\alpha}\not\leftarrow\widehat{x}Q & \to & (\textit{exp-outs}\not\leftarrow)\\ (\widehat{x}(P'\widehat{\alpha}\not\leftarrow\widehat{x}Q)\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{x}Q & \overset{\mathcal{X}}{=} & (3.5)\\ (\widehat{x}P'\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{x}Q & =_\alpha & (\widehat{x}P'\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}Q\end{array}$$

ii) If Q introduces x, we have two cases:
$Q=\langle x\cdot\beta\rangle$: By rule $(\not\rightarrow d)$.
$Q=Q_1\widehat{\beta}\,[x]\,\widehat{y}Q_2$: Then $x\notin fs(Q_1,Q_2)$, and

$$\begin{array}{lll}P'\widehat{\alpha}\not\rightarrow\widehat{x}(Q_1\widehat{\beta}\,[x]\,\widehat{y}Q_2) & \to & (\not\rightarrow\textit{imp-outs})\\ P'\widehat{\alpha}\dagger\widehat{v}((P'\widehat{\alpha}\not\rightarrow\widehat{x}Q_1)\widehat{\beta}\,[v]\,\widehat{y}(P'\widehat{\alpha}\not\rightarrow\widehat{x}Q_2)) & \overset{\mathcal{X}}{=} & (3.5)\\ P'\widehat{\alpha}\dagger\widehat{v}(Q_1\widehat{\beta}\,[v]\,\widehat{y}Q_2) & =_\alpha & P'\widehat{\alpha}\dagger\widehat{x}(Q_1\widehat{\beta}\,[x]\,\widehat{y}Q_2)\ \square\end{array}$$

Using coinduction, similar to the results above, we can also show:

Lemma 3.7 (GENERALISED RENAMING) *i*) $P\widehat{\alpha}\not\leftarrow\widehat{x}\langle x\cdot\beta\rangle \overset{\mathcal{X}}{=} P[\beta/\alpha]$.
ii) $\langle y\cdot\alpha\rangle\widehat{\alpha}\not\rightarrow\widehat{x}P \overset{\mathcal{X}}{=} P[y/x]$.

These results give that we can safely extend the notion of reduction on terms by adding:

$$\begin{array}{lrcll}(d\not\leftarrow): & P\widehat{\alpha}\not\leftarrow\widehat{x}Q & \to & P\widehat{\alpha}\dagger\widehat{x}Q & (P\ \textit{introduces}\ \alpha)\\ (d\not\leftarrow): & P\widehat{\alpha}\not\rightarrow\widehat{x}Q & \to & P\widehat{\alpha}\dagger\widehat{x}Q & (Q\ \textit{introduces}\ x)\\ (\not\leftarrow gc): & P\widehat{\alpha}\not\leftarrow\widehat{x}Q & \to & P & (\alpha\notin fp(P))\\ (gc\not\rightarrow): & P\widehat{\alpha}\not\rightarrow\widehat{x}Q & \to & Q & (x\notin fs(Q))\end{array}$$

thereby replacing the original deactivation rules, and $(cap\not\leftarrow)$ and $(\not\rightarrow cap)$.

Adding these more generic reduction rules gave another significant improvement on the efficiency of reduction, on which we report in Section 6.

4 A Term Graph Rewriting System for $\mathcal{X}$

As mentioned in the introduction, $\mathcal{X}$ corresponds to sequent calculus rather than natural deduction, leading to a reduction mechanism that is quite different from the applicative style of the λ-calculus. In fact, it is not specified through (implicit) substitution, but as a *conditional term rewriting system*; the non-standard aspect with respect to rewriting is a notion of *binding*.

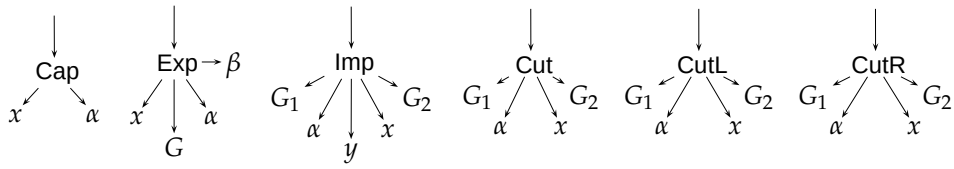
The choice to use term graph rewriting for the implementation of an interpreter of $\mathcal{X}$ was therefore the natural one: it provides the best platform for an efficient, elegant, and transparent implementation of a term rewriting system. The technique applied is the conventional one of [22, 6, 7], where terms and rewrite rules are *lifted* to graphs, and we use the standard *match*, *build*, *link*, *re-direct*, and *garbage collection* approach. By the process of lifting, the connectors appear only once in the generated graph, which immediately introduces sharing. Rewrite rules also become graphs with *two* sub-graphs that each possess a root, and are joined via shared leaves.

Term graphs are defined by:

Definition 4.1 (TERM GRAPHS) Using the signature $\{$ Cap, Exp, Imp, Cut, CutL, CutR$\}$, an infinite set of *graph variables* ranged over by $P, Q, R, \ldots$, an infinite set of *labels* ranged over by $k, l, m, n, \ldots$ and the set of *connector variables*, ranged over by $x, y, z, \ldots$ and $\alpha, \beta, \gamma, \ldots$, we define (ordered) graphs by the following grammar:

$$\begin{aligned} G \;::=\; & k : P \mid k :\ \mathsf{Cap}\,(l{:}x, m{:}\alpha) \mid k :\ \mathsf{Exp}\,(l{:}x, G, m{:}\alpha, n{:}\beta) \mid \\ & k :\ \mathsf{Imp}\,(G_1, l{:}\alpha, m{:}y, n{:}x, G_2) \mid k :\ \mathsf{Cut}\,(G_1, l{:}\alpha, m{:}x, G_2) \mid \\ & k :\ \mathsf{CutL}\,(G_1, l{:}\alpha, m{:}x, G_2) \mid k :\ \mathsf{CutR}\,(G_1, l{:}\alpha, m{:}x, G_2) \end{aligned}$$

which in graphs get represented by:



Since the graphs for left and right-activated cuts are much like unactivated cuts, we will only treat them separately when necessary.

In the second graph, for example, the connectors x, α and β could appear in G; then edges would point out from G to these connectors. We now give a formal definition of interpreting terms in $\mathcal{X}$ as graphs (using the notation of [9]).

Definition 4.2 (GRAPH INTERPRETATION) *i*) For each term, its *graph interpretation*, $\llbracket \cdot \rrbracket_{\mathrm{G}}$, expressed as a pair consisting of a label for the *root* of the graph and its *edges* and *sub-graphs* is inductively defined by in Figure 1.

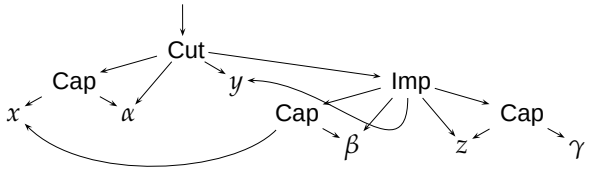
ii) We define the set of *initial* $\mathcal{X}$-graphs as the image of $\mathcal{X}$ terms under $\llbracket \cdot \rrbracket_{\mathrm{G}}$.

Notice that, by α-conversion, we can always assume that all bound connectors in a term have different names, and therefore will keep bound connectors separate when building the graph interpretation.

$$
\begin{array}{rcl}
\llbracket x \rrbracket_{\mathrm{G}} & = & \langle x \mid \emptyset \rangle \\
\llbracket \alpha \rrbracket_{\mathrm{G}} & = & \langle \alpha \mid \emptyset \rangle \\
\llbracket \langle x \cdot \alpha \rangle \rrbracket_{\mathrm{G}} & = & \langle r \mid \{r : \mathsf{Cap}(r_1, r_2)\} \cup G_1 \cup G_2 \rangle, \\
\text{where} & & \langle r_1 \mid G_1 \rangle \;=\; \llbracket x \rrbracket_{\mathrm{G}}, \\
& & \langle r_2 \mid G_2 \rangle \;=\; \llbracket \alpha \rrbracket_{\mathrm{G}} \\
\llbracket \widehat{y} P \widehat{\alpha} \cdot \beta \rrbracket_{\mathrm{G}} & = & \langle r \mid \{r : \mathsf{Exp}(r_1, r_2, r_3, r_4)\} \cup G_1 \cup G_2 \cup G_3 \cup G_4 \rangle \\
\text{where} & & \langle r_1 \mid G_1 \rangle \;=\; \llbracket \beta \rrbracket_{\mathrm{G}} \\
& & \langle r_2 \mid G_2 \rangle \;=\; \llbracket y \rrbracket_{\mathrm{G}} \\
& & \langle r_3 \mid G_3 \rangle \;=\; \llbracket P \rrbracket_{\mathrm{G}} \\
& & \langle r_4 \mid G_4 \rangle \;=\; \llbracket \alpha \rrbracket_{\mathrm{G}} \\
\llbracket P \widehat{\alpha} [y] \widehat{x} Q \rrbracket_{\mathrm{G}} & = & \langle r \mid \{r : \mathsf{Imp}(r_1, r_2, r_3, r_4, r_5)\} \\
& & \cup G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5 \rangle \\
\text{where} & & \langle r_1 \mid G_1 \rangle \;=\; \llbracket P \rrbracket_{\mathrm{G}} \\
& & \langle r_2 \mid G_2 \rangle \;=\; \llbracket \alpha \rrbracket_{\mathrm{G}} \\
& & \langle r_3 \mid G_3 \rangle \;=\; \llbracket y \rrbracket_{\mathrm{G}} \\
& & \langle r_4 \mid G_4 \rangle \;=\; \llbracket x \rrbracket_{\mathrm{G}} \\
& & \langle r_5 \mid G_5 \rangle \;=\; \llbracket Q \rrbracket_{\mathrm{G}} \\
\llbracket P \widehat{\alpha} \dagger \widehat{x} Q \rrbracket_{\mathrm{G}} & = & \langle r \mid \{r : \mathsf{Cut}(r_1, r_2, r_3, r_4)\} \cup G_1 \cup G_2 \cup G_3 \cup G_4 \rangle \\
\text{where} & & \langle r_1 \mid G_1 \rangle \;=\; \llbracket P \rrbracket_{\mathrm{G}} \\
& & \langle r_2 \mid G_2 \rangle \;=\; \llbracket \alpha \rrbracket_{\mathrm{G}} \\
& & \langle r_3 \mid G_3 \rangle \;=\; \llbracket x \rrbracket_{\mathrm{G}} \\
& & \langle r_4 \mid G_4 \rangle \;=\; \llbracket Q \rrbracket_{\mathrm{G}}
\end{array}
$$

Figure 1: Term graph interpretation of terms

Example 4.1 $\llbracket \langle x \cdot \alpha \rangle \widehat{\alpha} \dagger \widehat{y} (\langle x \cdot \beta \rangle \widehat{\beta} \; [y] \; \widehat{z} \langle z \cdot \gamma \rangle) \rrbracket_{\mathrm{G}}$ becomes



Definition 4.3 ($\mathcal{X}$-GRAPH REWRITING) *i*) The *lifting* of the reduction rules to term graph rewriting rules is expressed by first extending the interpretation of terms to graphs with the case for the (term) variables that occur in the rewrite rules:

$$\llbracket P \rrbracket_{\mathrm{G}} \;=\; \langle P \mid \emptyset \rangle$$

and then to define:

$$
\begin{array}{rcl}
\llbracket \mathit{left} \to \mathit{right} \rrbracket_{\mathrm{G}} & = & \langle r_l \mid G_l \cup G_r \rangle \\
\mathit{where} & & \langle r_l \mid G_l \rangle \;=\; \llbracket \mathit{left} \rrbracket_{\mathrm{G}} \\
& & \langle r_r \mid G_r \rangle \;=\; \llbracket \mathit{right} \rrbracket_{\mathrm{G}}
\end{array}
$$

Following [6, 7], these rules induce a notion $G \to_{\mathrm{G}} G'$ of term graph rewriting.

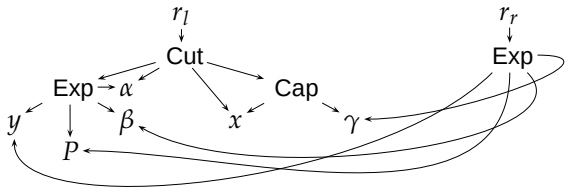
ii) We define the set of $\mathcal{X}$-graphs by closure under rewriting of initial $\mathcal{X}$-graphs.

Notice that since $\llbracket x \rrbracket_{\mathrm{G}} = \langle x \mid \emptyset \rangle$, and we can assume this to be unique, the left and right-hand side graphs for a rule are joined on the connectors.

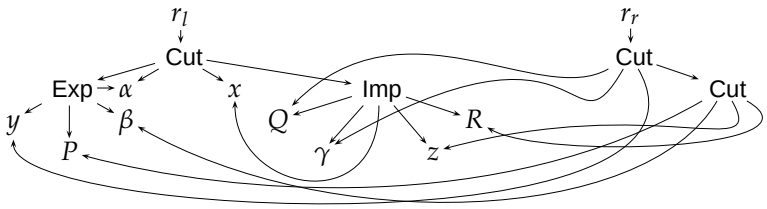
Example 4.2 We give some term graph rules[3]:

[3]Notice that it is perhaps more common to use $\perp$ for the nodes that can be matched against a graph or a connector node; we write the original names for readability and ease of definition of the interpretation function.

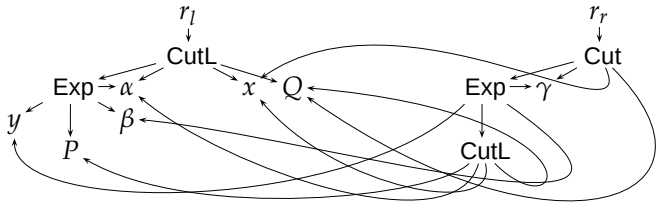
- $(exp):\ (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}\langle x\cdot\gamma\rangle\ \to\ \widehat{y}P\widehat{\beta}\cdot\gamma$



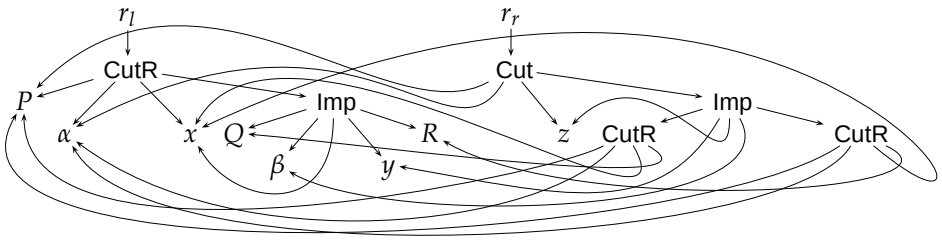
- $(exp\text{-}imp):\ (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}\,[x]\widehat{z}R)\ \to\ Q\widehat{\gamma}\dagger\widehat{y}(P\widehat{\beta}\dagger\widehat{z}R)$



- $(exp\text{-}outs\not\leftarrow):\ (\widehat{y}Q\widehat{\beta}\cdot\alpha)\widehat{\alpha}\not\leftarrow\widehat{x}P\ \to\ (\widehat{y}(Q\widehat{\alpha}\not\leftarrow\widehat{x}P)\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{x}P,\ (\gamma\ fresh)$



- $(\nwarrow imp\text{-}outs):\ P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\,[x]\widehat{y}R)\ \to\ P\widehat{\alpha}\dagger\widehat{z}((P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\beta}\,[z]\,\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R))$, with z fresh (with apologies for the spaghetti):



As can be seen, the amount of nodes added to the graph is small in comparison to the complexity of the graph generated by the rewriting; notice, for example, that an application of the third rule $(exp\text{-}outs\not\leftarrow)$ would only add the nodes containing Cut, Exp, γ, and CutL (that are accessible from the right-hand root r_r). Also, all edges coming into the node in the graph that is matched against the left-hand root r_l would be redirected into the new node Cut. The nodes containing CutL and Exp accessible from the left-hand root would become potential garbage.

Example 4.3 Take the reduction:

$$
\begin{array}{ll}
(\widehat{y}\langle y\cdot\beta\rangle\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{x}(\langle x\cdot\delta\rangle\widehat{\delta}\,[x]\,\widehat{v}\langle v\cdot\alpha\rangle) & \to^*\ (\nwarrow a),(\nwarrow imp\text{-}outs),(\nwarrow d),(exp)\\
(\widehat{y}\langle y\cdot\beta\rangle\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{z}((\widehat{y}\langle y\cdot\beta\rangle\widehat{\beta}\cdot\delta)\widehat{\delta}\,[z]\,\widehat{v}\langle v\cdot\alpha\rangle) & \to^*\ (exp\text{-}imp)\\
(\widehat{y}\langle y\cdot\beta\rangle\widehat{\beta}\cdot\sigma)\widehat{\sigma}\dagger\widehat{y}(\langle y\cdot\beta\rangle\widehat{\beta}\dagger\widehat{v}\langle v\cdot\alpha\rangle) & \to^*\ (cap)\\
(\widehat{y}\langle y\cdot\beta\rangle\widehat{\beta}\cdot\sigma)\widehat{\sigma}\dagger\widehat{y}\langle y\cdot\alpha\rangle & \to^*\ (exp)\\
\widehat{y}\langle y\cdot\beta\rangle\widehat{\beta}\cdot\alpha &
\end{array}
$$

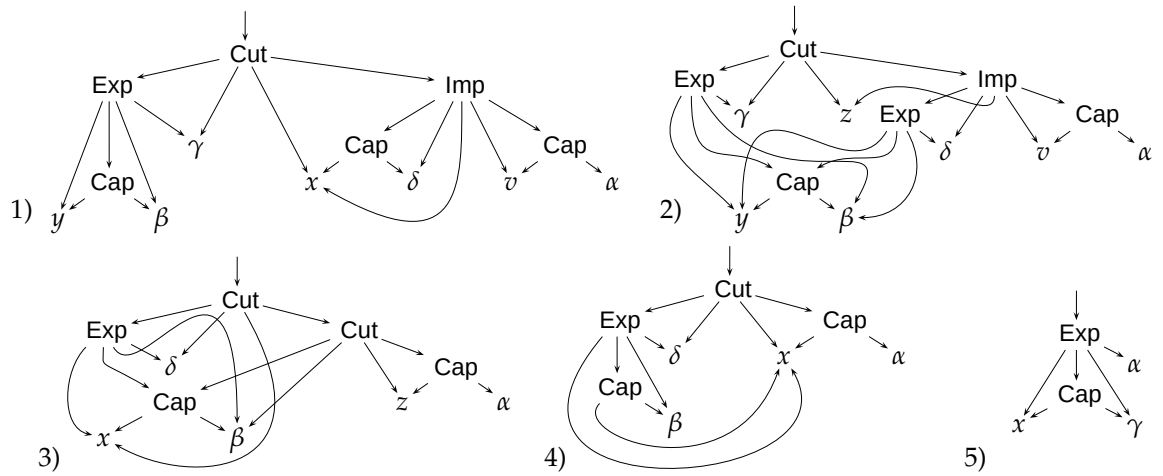


Figure 2: Graphs for Example 4.3

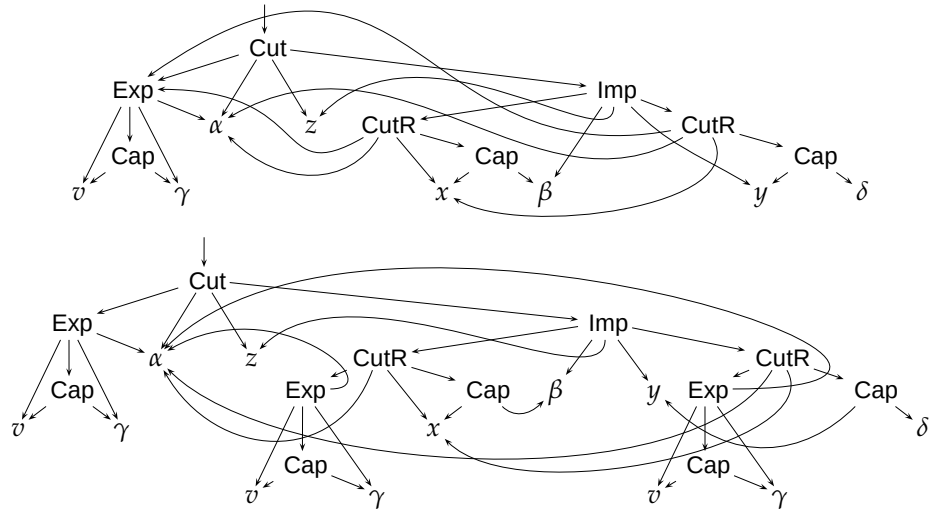


Figure 3: Unravelling a graph

The graph interpretation of this first term gives the first graph in Figure 2; reducing that graph using the corresponding term-graph rewriting rules generates the corresponding consecutive graphs.

In addition to the interpretation of terms to graphs, we would like an operation that transforms an $\mathcal{X}$-graph with sharing into one whose structure more closely resembles an $\mathcal{X}$-term. This is achieved by 'unravelling' the graph; copying out the shared nodes as far down as the connectors (which only appear once in a graph).

Definition 4.4 (CF [16]) *Unrv G*, the *unravelling* of a $\mathcal{X}$-graph G is obtained by traversing the graph top-down (notice that we have no cyclic structures), and copying, for all shared graphs, all nodes in that graph that are not free connectors.

Notice that both the set of *initial* $\mathcal{X}$-graphs and the image of the set of $\mathcal{X}$-graphs under *Unrv* are graphs containing sharing only at the level of connectors. This setup gives us a method of comparing an $\mathcal{X}$-term P with an $\mathcal{X}$-graph G, by comparing $\llbracket P \rrbracket_{\mathrm{G}}$ with *Unrv* G. This will be useful for formulating results later in the paper.

Example 4.4 Take G to be the first graph in Figure 3, then *Unrv* G is the second. Notice that the bound connectors v and γ within the shared graph

$$\mathsf{Exp}(1{:}v,\ \mathsf{Cap}(1,2),2{:}\gamma,\alpha)$$

are copied out, but α is not, and that the in-degree of α increases from four to six.

We now have the following results.

Lemma 4.5 If $G_1 \to_{\mathrm{G}} G_2$, then there exists G_3 such that Unrv $G_1 \to_{\mathrm{G}} G_3$, as well as Unrv $G_2 =$ Unrv G_3.

Proof: In each step of $G_1 \to_{\mathrm{G}} G_2$ a cut K is contracted. Using colouring, we can build a reduction sequence *Unrv* $G_1 \to_{\mathrm{G}} G_3$, for some G_3, by contracting, for each step in $G_1 \to_{\mathrm{G}} G_2$, always only all copies of K (using the same rule repeatedly). Notice that this reduction might have introduced sharing, and that G_2 and G_3 differ only in that G_3 contains less sharing than G_2, i.e. G_3 is a partially unravelled version of G_2. Since no other manipulation has been performed, we get *Unrv* $G_2 =$ *Unrv* G_3. □

We also have the following adequacy result:

Theorem 4.6 (ADEQUACY) *Let G_1, G_2 be $\mathcal{X}$-graphs, and P_1, P_2 be $\mathcal{X}$-terms such that Unrv $G_i = \mathbb{\llbracket} P_i \mathbb{\rrbracket}_{\mathrm{G}}$, for $i = 1,2$. If $G_1 \to_{\mathrm{G}} G_2$, then $P_1 \to P_2$. Moreover, if G_2 is in normal form, then so is P_2.*

Proof: By Lemma 4.5, we get that there exists a G_3 such that $\mathbb{\llbracket} P_1 \mathbb{\rrbracket}_{\mathrm{G}} \to_{\mathrm{G}} G_3$, as well as $\mathbb{\llbracket} P_2 \mathbb{\rrbracket}_{\mathrm{G}} =$ *Unrv* G_3. This reduction induces, similar to Lemma 4.5, a reduction from P_1 to P_2. If G_2 contains no cuts, then neither does $\mathbb{\llbracket} P_2 \mathbb{\rrbracket}_{\mathrm{G}}$, nor P_2. □

We can now prove the following result:

Theorem 4.7 *If $P \to Q$ in one step, then there exists a $\mathcal{X}$-graph G such that: $\mathbb{\llbracket} P \mathbb{\rrbracket}_{\mathrm{G}} \to_{\mathrm{G}} G$, and Unrv $G = \mathbb{\llbracket} Q \mathbb{\rrbracket}_{\mathrm{G}}$.*

Proof: Easy: in $\mathbb{\llbracket} P \mathbb{\rrbracket}_{\mathrm{G}}$, redexes are not shared; the only sharing in G is introduced by the reduction, which gets erased by unravelling. □

Notice that, by the non-confluent character for $\mathcal{X}$, we cannot prove a similar result for many-steps reduction paths, as illustrated by the following example.

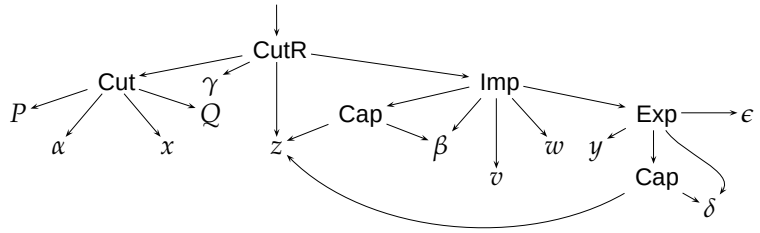
Example 4.5 Let P and Q be such that $\alpha \notin fp(P)$ and $x \notin fp(Q)$, so $P \leftarrow P\widehat{\alpha} \dagger \widehat{x}Q \to Q$. Now (assume $z \neq v$):

$$\begin{array}{ll}
(P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\gamma}\,\widehat{\backslash}\,\widehat{z}(\langle z{\cdot}\beta\rangle\,\widehat{\beta}\,[v]\ \widehat{w}(\widehat{y}\langle z{\cdot}\delta\rangle\,\widehat{\delta}{\cdot}\epsilon)) & \to\ (\backslash\mathit{imp\text{-}ins}),(\backslash\mathit{exp}),(\backslash d)(2\times) \\
((P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\beta\rangle)\,\widehat{\beta}\,[v]\ \widehat{w}(\widehat{y}((P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\delta\rangle)\,\widehat{\delta}{\cdot}\epsilon) & \\
\qquad\qquad \to^*\ (a\not{\prime}),(2.9),(\backslash a),(2.9) & \\
(P\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\beta\rangle)\,\widehat{\beta}\,[v]\ \widehat{w}(\widehat{y}(Q\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\delta\rangle)\,\widehat{\delta}{\cdot}\epsilon) &
\end{array}$$

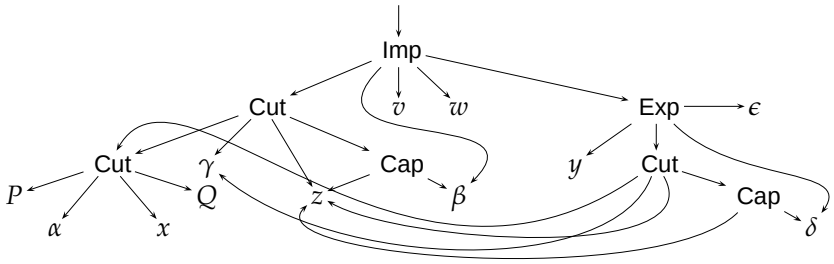
Notice that we have explicitly used the non-confluence of the cut $P\widehat{\alpha} \dagger \widehat{x}Q$, and reduced it once to P, and once to Q.

We cannot simulate this in the setting of Term Graph Rewriting. Instead, we get the follow-

ing graph for the first term



which, by $(\curlywedge imp\text{-}ins)$, $(\curlywedge exp)$, and $(\curlywedge d)(2\times)$ reduces to the graph



Of course, although shared, the cut $\mathsf{Cut}(P,\alpha,x,Q)$ can be reduced only *once*, resulting in either P or Q, implying that the above result cannot be achieved.

This is not unexpected, however, since all implementations of reduction systems will use a *reduction strategy*, preferring certain redexes over others, and thereby excluding other reduction paths; unfortunately, in a non-confluent setting, this means that certain reachable normal forms are no longer achievable.

5 Dealing with α-conversion in $\mathcal{X}$

In the rest of this paper we will discuss a number of solutions to the problem of capture avoidance / α-conversion in the context of $\mathcal{X}$. Capture avoidance is a well-known implementation issue; the most familiar context in which this problem occurs is perhaps the λ-calculus, where, when reducing a term like $(\lambda xy.xy)(\lambda xy.xy)$, α-conversion is essential. Without it, one would get

$$\begin{array}{rcll}(\lambda xy.xy)(\lambda xy.xy) & \to & (\lambda y.xy)[\lambda xy.xy/x] & \\ & = & \lambda y.(\lambda xy.xy)y & \\ & \to & \lambda y.(\lambda y.xy)[y/x] & \to\ \lambda yy.yy\end{array}$$

The conflict is caused by the fact that during the second reduction step, the free occurrence of y in $(\lambda xy.xy)y$ is brought *under* the binding λy, *i.e.* is *captured*.

A particular problem in dealing with α-conversion here is that the only reduction rule is $(\lambda x.M)N \to M[N/x]$, where the substitution is *implicit*, and supposed to be performed *immediately*. For example, when reducing

$$(\lambda xy.xy)(\lambda xy.xy) \to \lambda y.(\lambda xy.xy)y,$$

the latter term is *identical* to $\lambda y.xy[(\lambda xy.xy)/x]$. The actual performance of the substitution, which brings the right-most binder under the left-most is not part of the reduction system itself, but specified in the auxiliary definition of substitution.

A way to avoid this problem is, normally, to assume Barendregt's convention. In the reduction above, the first term satisfies this criterion, but the second does not: y is free as well as

bound in $(\lambda xy.xy)y$. Since this is not true for $(\lambda y.xy)[\lambda xy.xy/x]$, α-conversion must take place *during* the execution of substitution, and replace the bound name; hence, the result should be:

$$\begin{array}{lll}(\lambda xy.xy)(\lambda xy.xy) & \to & (\lambda y.xy)[\lambda xy.xy/x] \\ & = & \lambda y.(\lambda xz.xz)y \\ & \to & \lambda y.(\lambda z.xz)[y/x] \quad = \ \lambda yz.yz\end{array}$$

In the pure λ-calculus, it is impossible to deal with α-conversion since capturing cannot be expressed. The historical definition contained a notion of α-reduction in the language itself, but now normally the problem is avoided altogether by considering terms 'modulo α-conversion', and Barendregt's convention is cited, stating that bound and free names should be kept distinct by, when necessary, performing a silent α-conversion, thereby in fact leaving it to the implementer to treat.

For $\lambda\mathbf{x}$ the situation is slightly better, in that now substitution is an explicit part of term manipulation, so the creation of a naming conflict while reducing can be expressed. In fact, variable capturing can be avoided radically by changing the reduction rule

$$(\lambda y.M)\langle x:=P\rangle \to \lambda y.(M\langle x:=P\rangle)$$

into

$$(\lambda y.M)\langle x:=P\rangle \to \lambda z.(M\langle y:=z\rangle\langle x:=P\rangle), \quad (z \mathit{\ fresh})$$

thereby preventing a capture on a possibly bound y in P. This is expensive though, as it is performed on *all* substitutions on abstractions, and does not actually detect the capture, but just prevents it.

If one could use side-conditions on the rules as in λx_{gc} we could express that a free occurrence in a term will be captured, and define:

$$\begin{array}{lll}(\lambda y.M)\langle x:=P\rangle & \to & \lambda z.(M\langle y:=z\rangle\langle x:=P\rangle), \quad (z \mathit{\ fresh} \ \&\ y \in fs(P)) \\ (\lambda y.M)\langle x:=P\rangle & \to & \lambda z.(M\langle x:=P\rangle), \quad (y \notin fs(P))\end{array}$$

This observation lies at the basis of how we deal with α-conversion in Section 5.3.

Because it is an implementation issue, and we aim to have an efficient implementation of $\mathcal{X}$, we deal with the problem more formally. First we show that it is an issue to begin with.

Example 5.1 Take the term[4]

$$(\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\widehat{x}(\langle x\cdot\delta\rangle\,\widehat{\delta}\,[x]\ \widehat{w}\langle w\cdot\alpha\rangle)$$

which is also used in Example 4.3. Reducing this term differently, we get:

$$\begin{array}{lll}(\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\widehat{x}(\langle x\cdot\delta\rangle\,\widehat{\delta}\,[x]\ \widehat{w}\langle w\cdot\alpha\rangle) & \to^* & (\curlywedge a),(\curlywedge \mathit{imp\text{-}outs}) \\ (\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\widehat{z}(((\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\curlywedge\widehat{x}\langle y\cdot\mu\rangle)\,\widehat{\delta}\,[z]\ \widehat{w}((\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\curlywedge\widehat{x}\langle w\cdot\alpha\rangle)) & & \\ & \to^* & (\curlywedge d),(exp),(\curlywedge cap) \\ (\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\widehat{z}((\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\delta)\,\widehat{\delta}\,[z]\ \widehat{w}\langle w\cdot\alpha\rangle) & \to & (\mathit{exp\text{-}imp}) \\ (\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\delta)\,\widehat{\delta}\dagger\widehat{y}(\langle y\cdot\mu\rangle\,\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle) & \to^* & (\curlywedge a),(\curlywedge cut),(\curlywedge d),(\curlywedge cap) \\ ((\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\delta)\,\widehat{\delta}\dagger\widehat{y}\langle y\cdot\mu\rangle)\,\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \to & (exp) \\ (\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\mu)\,\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle & & \end{array}$$

[4]it is easy to check this term results from reducing the λ-term $xx\langle x:=\lambda y.y\rangle$; reducing this λ-term poses no α-conversion problem.

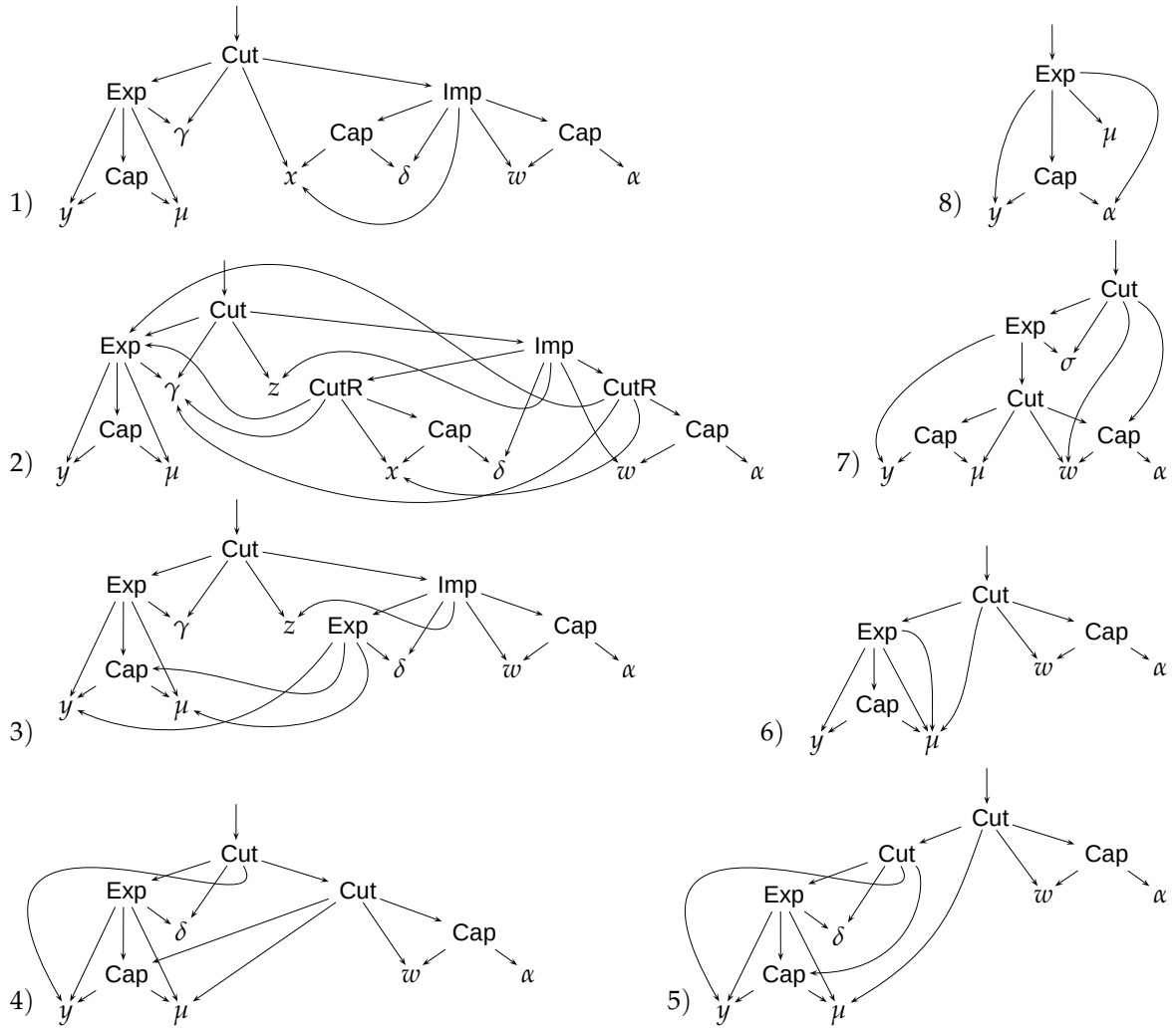


Figure 4: Graphs for Example 5.1

Notice that, in this term, μ is bound *twice*. To reduce the cut, we first check if μ is introduced; this is not so, since $\mu \in fp(\langle y \cdot \mu \rangle)$, thus denying the intended application of the rule (exp). Instead, we propagate the cut and obtain, using rules $(a\not\prec)$, $(exp\text{-}outs\not\prec)$ and $(d\not\prec)$,

$$\begin{array}{lll} (\widehat{y}\langle y \cdot \mu \rangle \widehat{\mu} \cdot \mu)\, \widehat{\mu} \dagger \widehat{w}\langle w \cdot \alpha \rangle & \to^* & (a\not\prec), (exp\text{-}outs\not\prec), (d\not\prec) \\ (\widehat{y}(\langle y \cdot \mu \rangle \widehat{\mu} \dagger \widehat{w}\langle w \cdot \alpha \rangle)\, \widehat{\mu} \cdot \sigma)\, \widehat{\sigma} \dagger \widehat{w}\langle w \cdot \alpha \rangle & \to^* & (cap), (exp) \\ \widehat{y}\langle y \cdot \alpha \rangle \widehat{\mu} \cdot \alpha & & \end{array}$$

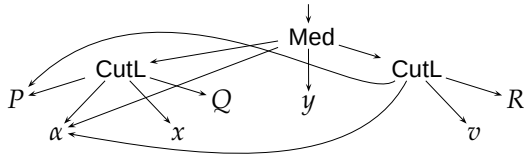
which is not the intended $\widehat{y}\langle y \cdot \mu \rangle \widehat{\mu} \cdot \alpha$. The problem is of course that we should have renamed one of the bound μ: this is an α-conversion problem, which is not solved by merely running the graph interpretation, as can be seen in Figure 4, where we show the corresponding graph reduction.

5.1 Lazy copying of shared graphs

The solution to the problem of capture we propose in this section is the one we first presented in [4]; it aims to avoid, as for *λ-graphs*, the sharing of graphs that are involved in more than one cut. Similarly to the case for the λ-calculus [29, 15], binding of connectors can be considered problematic in the context of sharing. Sharing an abstraction $\lambda x.G$ in *λ-graphs* is problematic

since the substitution is implemented via a redirection on G. This can be done only once, blocking a re-use of a shared abstraction, that therefore has to be copied first. To tackle this problem within the context of $\mathcal{X}$, a notion of *rebinding of sockets* and *of plugs* was introduced.

The basic idea is the following: suppose we are dealing with the graph



The fact that α is bound *twice* might cause the binders to become *nested* during computation. We avoid that by copying that part P that depends on α: we will 'peel off a copy' of the graphs which might get affected by the double binding of connectors.

This method is similar to that of [29], and differs in that we must copy several constructors and considering two classes of variables. An eager copy method will be too restrictive, in that it destroys a large amount of sharing. We will specify a lazy strategy that avoids much of the unnecessary copying in Section 6.3.

We extend the syntax of $\mathcal{X}$-graphs with two further constructors

$$P \ ::= \ \ldots \mid \mathsf{rp}(P,\alpha,\beta) \mid \mathsf{rs}(P,x,y)$$

that will represent the renaming of a bound (rebinding) plug or socket, respectively. This results in the (term graph) definition of rebinding a socket (rp) as given in Definition 5.1. These prevent a connector from being doubly bound by, essentially, copying that structure of a graph which contains that binder whilst introducing the new connector, thereby destroying the sharing of the connector via binding edges.

The function $\mathsf{rp}(P,\alpha,\beta)$ as given in Definition 5.1 is defined to build a new graph G' where the free occurrences of α in G are replaced with β and any binders encountered *in* G are made fresh. Since this is, essentially, a copying function, when we move the rebinding mechanism *under* binders, as in the third case below, we would create double binders for those bound connectors we have just passed. Therefore, we need to rebind those as well.

Definition 5.1 (REBINDING REWRITE RULES) The function rp is defined by the term graph rewriting rules in Figure 5.

$$\begin{array}{lrcl}
(rpGC) & \mathsf{rp}(P,\beta,\gamma) & \to & P, \qquad \beta \text{ not free in } P \\
(rpCapRen) & \mathsf{rp}(\langle x\cdot\beta\rangle,\beta,\gamma) & \to & \langle x\cdot\gamma\rangle \\
(rpExp) & \mathsf{rp}(\widehat{y}P\widehat{\alpha}\cdot\eta,\beta,\gamma) & \to & \widehat{k}\ \mathsf{rs}(\mathsf{rp}(\mathsf{rp}(P,\beta,\gamma),\alpha,\lambda),y,k)\ \widehat{\lambda}\cdot\eta, \quad (\eta \neq \beta) \\
(rpExpRen) & \mathsf{rp}(\widehat{y}P\widehat{\alpha}\cdot\beta,\beta,\gamma) & \to & \widehat{k}\ \mathsf{rs}(\mathsf{rp}(\mathsf{rp}(P,\beta,\gamma),\alpha,\lambda),y,k)\ \widehat{\lambda}\cdot\gamma \\
(rpMed) & \mathsf{rp}(P\widehat{\alpha}\,[x]\,\widehat{y}Q,\beta,\gamma) & \to & \mathsf{rp}(\mathsf{rp}(P,\beta,\gamma),\alpha,\eta)\ \widehat{\eta}\ [x]\ \widehat{z}\ \mathsf{rs}(\mathsf{rp}(Q,\beta,\gamma),y,z) \\
(rpCut) & \mathsf{rp}(P\widehat{\alpha}\dagger\widehat{y}Q,\beta,\gamma) & \to & \mathsf{rp}(\mathsf{rp}(P,\beta,\gamma),\alpha,\eta)\ \widehat{\eta}\dagger\widehat{z}\ \mathsf{rs}(\mathsf{rp}(Q,\beta,\gamma),y,z)
\end{array}$$

(The function rs is defined similarly.) Notice that the call to the function rp builds a version of P that uses a fresh socket γ to connect rather than β. Also, all bound connectors are renamed: evaluating the rebinding rules builds a version of P with fresh binder names. This ensures there is only ever one pointer to nodes that bind over P or the local binders in P. This comes, however, at the price of a large rebinding overhead.

The functions rs and rp are expressed as higher-order term-graph rewriting rules. Because these higher-order functions may not necessarily be evaluated eagerly, they may interfere

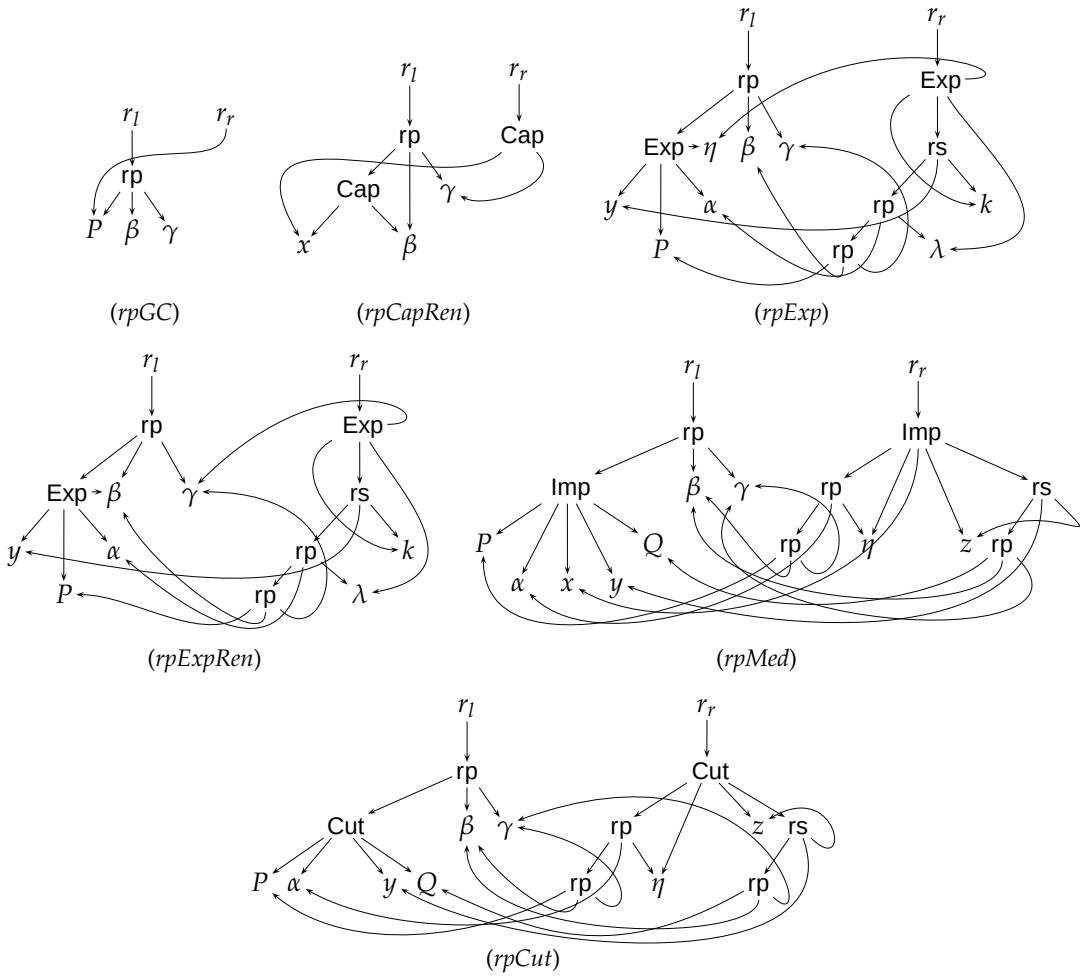


Figure 5: Rebinding plugs

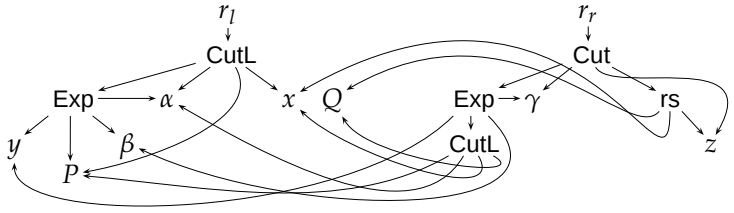
with the reductions of the $\mathcal{X}$-calculus: if the sub-circuit of an inactive cut is a rebinding term, an activation will be forced even though the sub-circuit of the rebinding term introduces the appropriate connector of the cut (and a logical rule should therefore have be applied).

Rather than forcing the evaluation of these rebinding constructs to completion via an 'eager' reduction strategy, we will define a lazier evaluation strategy that avoids this mis-activation in Section 6.3.

Using the functions rs and rp gives a different formal definition of interpreting rewrite rules in $\mathcal{X}$ as graphs. As suggested by the example above (Example 5.1), term rewrite rules which introduce sharing of binders need to copy these in order to avoid capture.

Definition 5.2 (COPYING TGRS REWRITE RULES) **Left propagation**

$$(\textit{exp-outs}\not\cdot)\colon\ (\widehat{y}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\not\cdot\widehat{x}Q\ \to\ (\widehat{y}(P\widehat{\alpha}\not\cdot\widehat{x}Q)\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{z}\mathsf{rs}(Q,x,z)$$



$(\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\,\widehat{x}(\langle x\cdot\delta\rangle\,\widehat{\delta}\,[x]\;\widehat{w}\langle w\cdot\alpha\rangle)$ $\quad\to^*\ (\curlywedge a),(\curlywedge \textit{imp-outs})$

$(\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\,\widehat{z}((\mathsf{rp}((\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma),\gamma,\tau)\,\widehat{\tau}\curlywedge\widehat{x}\langle x\cdot\delta\rangle)\,\widehat{\beta}\;[z]\;\widehat{y}(\mathsf{rp}(\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma,\gamma,\sigma)\,\widehat{\sigma}\curlywedge\widehat{x}\langle w\cdot\alpha\rangle))\ \to^*(\mathsf{rp})$

$(\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\,\widehat{z}(((\widehat{v}\langle v\cdot\nu\rangle\,\widehat{\nu}\cdot\tau)\,\widehat{\tau}\curlywedge\widehat{x}\langle x\cdot\delta\rangle)\,\widehat{\delta}\;[z]\;\widehat{w}((\widehat{u}\langle u\cdot\eta\rangle\,\widehat{\eta}\cdot\sigma)\,\widehat{\sigma}\curlywedge\widehat{x}\langle w\cdot\alpha\rangle)))$

$\quad\to^*\ (\curlywedge d),(\textit{exp}),(\curlywedge \textit{cap})$

$(\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\,\widehat{z}((\widehat{v}\langle v\cdot\nu\rangle\,\widehat{\nu}\cdot\delta)\,\widehat{\delta}\;[z]\;\widehat{w}\langle w\cdot\alpha\rangle)$ $\quad\to\ (\textit{exp-imp})$

$(\widehat{v}\langle v\cdot\nu\rangle\,\widehat{\nu}\cdot\delta)\,\widehat{\delta}\dagger\,\widehat{y}(\langle y\cdot\mu\rangle\,\widehat{\mu}\dagger\,\widehat{w}\langle w\cdot\alpha\rangle)$ $\quad\to^*\ (\curlywedge a),(\curlywedge \textit{cut}),(\curlywedge \textit{exp}),(\curlywedge d)$

$((\widehat{v}\langle v\cdot\nu\rangle\,\widehat{\nu}\cdot\delta)\,\widehat{\delta}\dagger\,\widehat{y}\langle y\cdot\mu\rangle)\,\widehat{\mu}\dagger\,\widehat{w}\langle w\cdot\alpha\rangle$

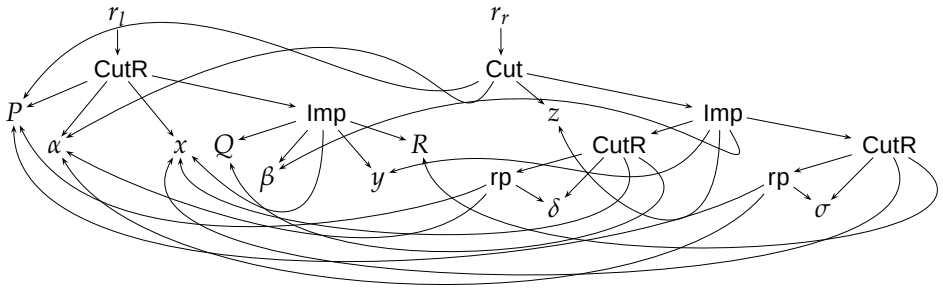
Figure 6: Correction using rebinding of Example 5.1.

$$(\textit{imp}\curlywedge):\ (Q\widehat{\beta}[z]\,\widehat{y}R)\,\widehat{\alpha}\curlywedge\widehat{x}P\ \to\ (Q\widehat{\alpha}\curlywedge\widehat{x}P)\,\widehat{\beta}\;[z]\;\widehat{y}(R\widehat{\alpha}\curlywedge\widehat{k}\,\mathsf{rs}(P,x,k))$$

$$(\textit{cut}\curlywedge):\ (Q\widehat{\beta}\dagger\,\widehat{y}R)\,\widehat{\alpha}\curlywedge\widehat{x}P\ \to\ (Q\widehat{\alpha}\curlywedge\widehat{x}P)\,\widehat{\beta}\dagger\,\widehat{y}(R\widehat{\alpha}\curlywedge\widehat{k}\,\mathsf{rs}(P,x,k))$$

Right propagation

$$(\curlywedge \textit{imp-outs}):\ P\widehat{\alpha}\curlywedge\widehat{x}(Q\widehat{\beta}[x]\,\widehat{y}R)\ \to\ P\widehat{\alpha}\dagger\,\widehat{z}((\mathsf{rp}(P,\alpha,\mu)\,\widehat{\mu}\curlywedge\widehat{x}Q)\,\widehat{\beta}\;[z]\;\widehat{y}(\mathsf{rp}(P,\alpha,\eta)\,\widehat{\eta}\curlywedge\widehat{x}R))$$



$$(\curlywedge \textit{imp-ins}):\ P\widehat{\alpha}\curlywedge\widehat{x}(Q\widehat{\beta}[z]\,\widehat{y}R)\ \to\ (P\widehat{\alpha}\curlywedge\widehat{x}Q)\,\widehat{\beta}\;[z]\;\widehat{y}(\mathsf{rp}(P,\alpha,\gamma)\,\widehat{\gamma}\curlywedge\widehat{x}R)$$

$$(\curlywedge \textit{cut}):\ P\widehat{\alpha}\curlywedge\widehat{x}(Q\widehat{\beta}\dagger\,\widehat{y}R)\ \to\ (P\widehat{\alpha}\curlywedge\widehat{x}Q)\,\widehat{\beta}\dagger\,\widehat{y}(\mathsf{rp}(P,\alpha,\gamma)\,\widehat{\gamma}\curlywedge\widehat{x}R)$$

That this (partial) copy action gives a solution to the name capture problem is perhaps not obvious. In fact, running the modified system on terms that have a bound connector occur more than once would not guaranteed to be safe. However, if the initial term has different names for all bound connectors, which in turn are different from all free names, then this property is preserved during reduction. So it is impossible for names to be captured.

Example 5.2 A corrected reduction, using rebinding, for Example 5.1 can be found in Figure 6. Notice that this time there is no possibility of variable clash, since there are no shared binders (the other copy of the μ binder together with the μ in the capsule is renamed to ν). The highlighted cut $\widehat{\mu}\dagger\,\widehat{w}$ can be activated and safely propagated through the left sub-circuit.

The solution using rebinding is surprisingly easy to formulate, and only the rules that use explicit replication need to be changed, but comes at the price of having to extend the signature of the calculus, as well as the set of rewrite rules. Moreover, it turns out to be highly inefficient; this is of course mainly due to the loss of sharing. The main objection to rebinding is that it creates unnecessary overhead in that it invokes rebinding for all double bindings of connectors, regardless of whether or not they created a conflict; as we will see in the final section, the cost of rebinding is high.

5.2 Preserving Barendregt's convention

Although the solution of the previous section is correct in that it avoids the creation of nested bindings, the implementation was slow, mainly due to the large amount of extra reduction that were introduced as a result of the copying process, even when done lazily.

More importantly, we observed that, using term graph rewriting techniques, the particular λ-graph problem disappears; substitution on a graph G is now specified *explicitly* by reduction rules, and application of these rules will have the effect that a term is generated *of the same shape as* G, to which the substitution has been applied. In fact, a large part of the necessary copying action is done during the *build* phase of the rewrite mechanism, and does not need to be treated at the level of the reduction rules. The conclusion of this was that, although the copying solved the α-conversion problem, here the problem is of a different nature, i.e. that of *capturing*.

In this section, we will propose a solution for the α-conversion problem in $\mathcal{X}$, by preserving Barendregt's convention on names, i.e. make sure that names never occur both free and bound. We will achieve this by detecting and avoiding name clashes, without having to extend the syntax of the calculus; we just modify the (side-conditions of) the rules.

To tackle it in a formal way, we first introduce the notion of α-safety.

Definition 5.3 (α-SAFETY) We call a *term* ($\mathcal{X}$-graph) *α-safe* if it adheres to Barendregt's convention, i.e.: (1) no connector occurs both free and bound, and (2) no nesting of binders to the same connector occurs. We call a *rewrite rule α-safe* if it respects α-safety, that is, it rewrites an α-safe term (graph) to an α-safe term (graph). We call a rewrite system α-safe if all its rules are α-safe.

Example 5.4 The term $(\widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}\cdot\mu)\,\widehat{\mu}\dagger\widehat{w}\langle w{\cdot}\alpha\rangle$ is not α-safe (it fails both criteria); neither is $(\widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}\cdot\delta)\,\widehat{\mu}\dagger\widehat{w}\langle w{\cdot}\alpha\rangle$, by the first criterion.

In order to obtain an α-safe implementation of $\mathcal{X}$, we need to identify the rewrite rules that are *not* α-safe.

Example 5.5 In Example 5.1, the application of (*exp-imp*) in Step 3 violates our α-safety property since μ is both bound and free in $(\widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}\cdot\delta)\,\widehat{\delta}\dagger\widehat{y}\langle y{\cdot}\mu\rangle$. So rule (*exp-imp*) is *not* α-safe; for the left-hand side $(\widehat{y}P\,\widehat{\beta}\cdot\alpha)\,\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}R)$ to be α-safe, the connectors y and z, and β and γ are allowed to be the same: they are not nested. However, this is no longer true for the right-hand sides: y and z occur nested in the first alternative $Q\widehat{\gamma}\dagger\widehat{y}(P\widehat{\beta}\dagger\widehat{z}R)$, and β and γ in the second $(Q\widehat{\gamma}\dagger\widehat{y}P)\,\widehat{\beta}\dagger\widehat{z}R$, which might force an α-conversion to be necessary.

Since we do not necessarily need to avoid connectors being bound twice (as long as they are not nested), in dealing with the necessary renaming of bound connectors we can take advantage of the explicit renaming feature of $\mathcal{X}$, using new cuts such as $\langle v{\cdot}\delta\rangle\,\widehat{\delta}\nwarrow\widehat{y}P$ or $P\,\widehat{\beta}\nearrow\widehat{v}\langle v{\cdot}\delta\rangle$ to rename y by v, or β by δ respectively in P, where v,δ are fresh (see Lemma 2.9). By activating the cuts, the renaming is forced to take place and other cuts are prohibited from propagating over the renaming.

On the down-side, unlike for the rebinding solution where we added extra nodes to our graphs to express the copying, it is no longer possible to force eager or lazy evaluation of α-conversion without doing the same with the general propagation rules.

We discussed above in Example 5.5 that we should perform the α-conversion in rule (*exp-imp*). Let us consider the first choice (assume α and x are introduced); remember that y might be z or appear bound in P without violation of Barendregt's convention:

$$(\widehat{y}R\,\widehat{\beta}\cdot\alpha)\,\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P)\;\rightarrow\;Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\dagger\widehat{z}P)$$

If we assume that the α-safety criterion holds on an instance of the left-hand side of the rule (so $\gamma \notin bp(\widehat{y}P\widehat{\beta}\cdot\alpha)$ and $y \notin bs(Q\widehat{\gamma}[x]\widehat{z}R)$), then the application of the rule should not break the α-safety criterion. We require the right-hand side of the rule to also be α-safe. Notice that in its current form, the rule does indeed break the criterion, since β and γ are nested on the right-hand side and the left-hand side places no constraints on the relation between β and γ. A term graph on which this rule is applied may have $\beta{=}\gamma$, in which case the application of the rule will have created a nested binding.

Since now we do not necessarily need to avoid connectors being bound twice (as long as they are not nested), we do not need to completely copy terms. Instead, in dealing with the necessary renaming of bound connectors we can take advantage of the explicit renaming feature of $\mathcal{X}$, introducing to the rules new cuts such as $\langle v\cdot\delta\rangle\,\widehat{\delta}\nwarrow\widehat{y}P$ or $P\widehat{\beta}\nearrow\widehat{v}\langle v\cdot\delta\rangle$ to rename y by v, or β by δ respectively in P, where v,δ are fresh (see Lemma 2.9). By activating the cuts, the intention is to force the renaming to take place first. We will also need to adopt our proposed strongly normalising rules, which prevent a cut from deactivating, thereby enforcing priority to the renaming cuts.

Returning to the violation in the $(\textit{exp-imp}_{\mathrm{N}})$ rule, in order to ensure the rewrite will be executed correctly (with respect to α-safety), we need to introduce an extra constraint to the applicability of the rule $(\textit{exp-imp}_{\mathrm{N}})$, namely $\gamma \notin bs(\widehat{y}P\widehat{\beta}\cdot\alpha)$. (This can be equivalently formulated as $\gamma \notin bs(P) \wedge \beta \neq \gamma$.) If the side-condition does not hold, then applying the rule will create a nested binding of (the image of) γ in the term graph. To remedy the situation, we must pay the cost of a renaming. This implies that there are now two alternatives for this rule:

$$(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) \;\to\; \begin{cases} Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\dagger\widehat{z}P) & (y\neq z, y\notin bs(P)) \\ Q\widehat{\gamma}\dagger\widehat{v}((\langle v\cdot\delta\rangle\widehat{\delta}\nwarrow\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P) & \\ & (y=z\vee y\in bs(P), v,\delta\ \textit{fresh}) \end{cases}$$

Likewise, there are two alternatives for the second choice:

$$(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) \;\to\; \begin{cases} (Q\widehat{\gamma}\dagger\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P & (\gamma\neq\beta\notin bp(Q)) \\ (Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\nearrow\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{z}P & \\ & (\beta=\gamma\vee\beta\in bp(Q), v,\delta\ \textit{fresh}) \end{cases}$$

Example 5.6 Applying this solution to Example 5.1, we have, instead of the problematic step

$$\begin{array}{l}(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\gamma)\widehat{\gamma}\dagger\widehat{k}((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\,[k]\;\widehat{w}\langle w\cdot\alpha\rangle) \;\to\; (\textit{exp-imp}) \\ ((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}\langle y\cdot\mu\rangle)\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle\end{array}$$

the correction

$$\begin{array}{ll}(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\gamma)\widehat{\gamma}\dagger\widehat{k}((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\,[k]\;\widehat{w}\langle w\cdot\alpha\rangle) & \to\; (\textit{exp-imp}) \\ ((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}(\langle y\cdot\mu\rangle\widehat{\mu}\nearrow\widehat{v}\langle v\cdot\beta\rangle))\widehat{\beta}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \to\; (d\nearrow,\cap) \\ ((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}\langle y\cdot\beta\rangle)\widehat{\beta}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \to\; (\textit{exp}) \\ (\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\beta)\widehat{\beta}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \to\; (\textit{exp}) \\ \widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\alpha & \end{array}$$

To guarantee α-safety, we need to make a similar change to each rule where a possible α-conflict is introduced, like $(\nwarrow\textit{cut})$:

$$P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) \;\to\; (P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\beta}\dagger\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R)$$

There are two points of concern here: $\alpha=\beta$, and β or y occurs bound in P (notice that $x\neq y$ as, by assumption, the left-hand side is an α-safe term). With this in mind, the rule $(\nwarrow\textit{cut})$

is amended with extra side conditions and replaced by the following variants (where v, δ are *fresh*):

$$P\widehat{\alpha}\mathbin{\backslash}\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) \rightarrow \begin{cases} (P\widehat{\alpha}\mathbin{\backslash}\widehat{x}Q)\widehat{\beta}\dagger\widehat{y}(P\widehat{\alpha}\mathbin{\backslash}\widehat{x}R) & (\beta\notin bp(P)\ \&\ y\notin bs(P)) \\ (P\widehat{\alpha}\mathbin{\backslash}\widehat{x}Q)\widehat{\beta}\dagger\widehat{v}(P\widehat{\alpha}\mathbin{\backslash}\widehat{x}(\langle v\cdot\delta\rangle\widehat{\delta}\mathbin{\backslash}\widehat{y}R)) & (\beta\notin bp(P)\ \&\ y\in bs(P)) \\ (P\widehat{\alpha}\mathbin{\backslash}\widehat{x}(Q\widehat{\beta}\not\vdash\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{y}(P\widehat{\alpha}\mathbin{\backslash}\widehat{x}R) & (\beta\in bp(P)\ \&\ y\notin bs(P)) \\ (P\widehat{\alpha}\mathbin{\backslash}\widehat{x}(Q\widehat{\beta}\not\vdash\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{v}(P\widehat{\alpha}\mathbin{\backslash}\widehat{x}(\langle v\cdot\delta\rangle\widehat{\delta}\mathbin{\backslash}\widehat{y}R)) & (\beta\in bp(P)\ \&\ y\in bs(P)) \end{cases}$$

Almost all propagation rules (except for $(d\not\vdash)$, $(cap\not\vdash)$, $(\backslash d)$, and $(\backslash cap)$) should be treated like this; of the logical rules, only rule $(exp\text{-}imp)$ needs dealing with as specified above. In all, this gives a much more complicated rewriting system, with a great many rewrite rules. The advantage of this approach is that α-conversion itself is detected and dealt with:

Theorem 5.7 *Let $\rightarrow_\alpha$ stand for the notion of rewriting on $\mathcal{X}$ obtained by changing the rules as above. Then: if P, Q are pure terms such that P is α-safe, and $P\rightarrow_\alpha Q$, then Q is α-safe.*

Proof: Straightforward, by verifying that each individual rewriting step preserves α-safeness. □

The computational cost of this approach is low compared to the rebinding approach; the price to pay is an increase in the number of rules. However, the detection of a possible 'α-danger' in a rule is straightforward, since it only depends on the information present in the rule, so it is even possible to, at the user level, allow for the definition of the normal rules, and to automatically generate the α-safe variants.

5.3 Avoiding capture

Barendregt's convention is a perfectly adequate solution to the α-conversion problem, since it forbids a term with nested binders to the same variable to be created, thereby totally avoiding any ambiguity to the system. However, one can justifiably argue that the convention is restrictive, and expensive to uphold at run-time. After all, allowing nesting of bound variables as in $\lambda y.(\lambda xy.xy)y$ (and therefore also of variables occurring free and bound variables as in $(\lambda xy.xy)y$) need not be ambiguous at all when considering the *innermost* of nested binders the strongest; in this paradigm, the only thing that needs to be avoided during reduction is that of *capture* of free connectors, bringing a connector under a binder.

Example 5.8 Referring back to Example 5.1, recognising that μ is bound twice, we would like to apply rule (exp) to the term

$$(\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\mu)\,\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle$$

But since that rule checks if μ occurs free in $\langle y\cdot\mu\rangle$, which it does, we are forced to left-activate the term, which eventually gives the wrong result. However, once we accept that we only need to check that a connector does not get captured - notice that this is not the case here - and that we allow connectors to appear both free and bound, we only need to check that those free occurrences of α are indeed those that we would associate with the free α on the outside of the export term. So, instead of applying rule (exp),

$$(\widehat{x}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{y}\langle y\cdot\gamma\rangle \;\rightarrow\; \widehat{x}P\widehat{\beta}\cdot\gamma \quad (\alpha\notin fp(P))$$

we should apply the variant

$$(\widehat{x}P\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{y}\langle y\cdot\gamma\rangle \;\rightarrow\; \widehat{x}P\widehat{\alpha}\cdot\gamma \quad (\alpha=\beta\vee\alpha\notin fp(P))$$

Then the reduction of Example 5.1 should contain:

$$(\widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}{\cdot}\mu)\,\widehat{\mu}\dagger\widehat{w}\langle w{\cdot}\alpha\rangle \;\to\; \widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}{\cdot}\alpha$$

We will show that we can identify the situations in which the application of a rule results in capturing, and will modify the rules in such a way that the necessary α-conversions are automatically preformed. The solution is, in appearance, strikingly similar to that of Section 5.2 but for the fact that *freeness* is used rather than *boundness*. In Section 6.6, this approach will be shown to be much more efficient; this is mainly because the solution of Section 5.2, many terms which are not 'α-safe' (Definition 5.3) are left untouched here.

Example 5.9 Let us consider the rule (*exp-imp*$_{\mathrm{N}}$):

$$(\widehat{y}R\widehat{\beta}{\cdot}\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) \;\to\; (Q\widehat{\gamma}\dagger\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P\alpha, \quad (x \textit{ introduced})$$

In order to allow the rewrite to be executed like this, the side condition should express an extra criterion to avoid the capture of a free β in Q; if $\beta{\in}fs(Q)$, then the rule would bring that β under the binder $\widehat{y}$ on the right-hand side, and renaming should take place. Also, notice that if $\beta{=}\gamma$, there would be no capture, since the order of nested binders are preserved. This implies that there are now two alternatives for the rule (*exp-imp*$_{\mathrm{N}}$). We define these respectively as:

$$(\widehat{y}R\widehat{\beta}{\cdot}\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) \;\to\; \begin{cases} (Q\widehat{\gamma}\dagger\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P & (\beta\notin fp(Q)\vee\beta=\gamma)\\ (Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\not\leftarrow\widehat{v}\langle v{\cdot}\delta\rangle))\widehat{\delta}\dagger\widehat{z}P & \\ & (\beta\in fp(Q)\wedge\beta\neq\gamma)\end{cases}$$

where v,δ are fresh. Likewise, the respective rules for the CBV variant are:

$$(\widehat{y}R\widehat{\beta}{\cdot}\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) \;\to\; \begin{cases} Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\dagger\widehat{z}P) & (y\notin fs(P)\vee y=z)\\ Q\widehat{\gamma}\dagger\widehat{v}((\langle v{\cdot}\delta\rangle\widehat{\delta}\nwarrow\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P) & \\ & (y\in fs(P)\wedge y\neq z)\end{cases}$$

where v,δ are fresh.

Also, since now we explicitly allow for connectors to occur both free and bound in a term, the rules need to check if the connector we try to connect to in a cut is actually really free.

For example, rule (*exp-outs*$\not\leftarrow$) now becomes:

$$(\widehat{y}Q\widehat{\beta}{\cdot}\alpha)\widehat{\alpha}\not\leftarrow\widehat{x}P \;\to\; \begin{cases} (\widehat{y}(Q\widehat{\alpha}\not\leftarrow\widehat{x}P)\widehat{\beta}{\cdot}\gamma)\widehat{\gamma}\dagger\widehat{x}P, & (\alpha\in fp(Q)\;\&\;\alpha\neq\beta)\\ (\widehat{y}Q\widehat{\beta}{\cdot}\alpha)\widehat{\alpha}\dagger\widehat{x}P, & (\alpha\notin fp(Q)\vee\alpha=\beta)\end{cases}$$

Rule ($\nwarrow$*cut*) becomes:

$$P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R)\to \begin{cases} (P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\beta}\dagger\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R) & (y\notin fs(P)\wedge y\neq x\wedge\beta\notin fp(P)\wedge\beta\neq\alpha)\\ (P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\beta}\dagger\widehat{v}(P\widehat{\alpha}\nwarrow\widehat{x}(\langle v{\cdot}\delta\rangle\widehat{\delta}\nwarrow\widehat{y}R)) & \\ & (y\notin fs(P)\wedge y\neq x\wedge(\beta\in fp(P)\vee\beta=\alpha))\\ (P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\not\leftarrow\widehat{v}\langle v{\cdot}\delta\rangle))\widehat{\delta}\dagger\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R) & \\ & ((y\in fs(P)\vee y=x)\wedge\beta\notin fp(P)\wedge\beta\neq\alpha)\\ (P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\not\leftarrow\widehat{v}\langle v{\cdot}\delta\rangle))\widehat{\delta}\dagger\widehat{v}(P\widehat{\alpha}\nwarrow\widehat{x}(\langle v{\cdot}\delta\rangle\widehat{\delta}\nwarrow\widehat{y}R)) & \\ & (y\in fs(P)\vee y=x\vee\beta\in fp(P)\vee\beta=\alpha)\end{cases}$$

All the rules need to be modified to check for possible capture of connectors. Although the structure of these new rules is similar to those in Section 5.2, the improvement in execution speed is impressive, as can be seen in the last section.

6 Reduction Strategies for $\mathcal{X}$

In Section 5, we studied different schemes for avoiding capture in the context of the $\mathcal{X}$-calculus. In this section, we compare the cost of each scheme when used to reduce expressions, and obtain a cost of the renaming operations required to uphold a particular α-safety criteria; e.g. in the 'preserving Barendregt's convention' scheme, the cost refers to the rewrite operations that must be carried out to ensure no nested binding is created during a reduction.

To allow for a fair and accurate comparison across the schemes we proposed, it is important that, aside from the α-conversion steps, the same reduction paths are chosen when evaluating a term. This implies the need for a *deterministic* reduction strategy. Furthermore, this strategy should not be affected by any renaming cuts performing α-conversions. The following example shows how a naive reduction strategy (in the call-by-name subsystem) could be affected, and motivates the need for an extension to the $\mathcal{X}$ implementation to allow for complex strategies to be defined.

Example 6.1 Consider an instance of a graph where the rule (insRrenP) from the capture avoidance scheme (Section 5.3) is applicable. Let us assume the subgraphs Q and R are pure and R introduces y. We have the following reduction.

$$\begin{array}{ll} 1. & (\widehat{y} R \widehat{\beta}\cdot\alpha)\,\widehat{\alpha} \dagger \widehat{x}(\langle v\cdot\beta\rangle\,\widehat{\delta}\,[x]\,\widehat{z}Q) \quad \rightarrow \\ 2. & (\langle v\cdot\beta\rangle\,\widehat{\delta} \dagger \widehat{y}(R\,\widehat{\beta} \nearrow \widehat{f}\,\langle f\cdot\sigma\rangle))\,\widehat{\sigma} \dagger \widehat{z}Q \end{array}$$

Note that if the α-conversion steps had been instantaneous, we would have obtained:

$$2'. \quad (\langle v\cdot\beta\rangle\,\widehat{\delta} \dagger \widehat{y}R')\,\widehat{v} \dagger \widehat{z}Q$$

(where $R' = R[\sigma/\beta]$). In Step 2, although R introduces y, we cannot evaluate the cut $\widehat{\delta} \dagger \widehat{y}$ without activating it first, which will cause it to (wastefully) propagate through R' (once the renaming cut $\widehat{\beta} \nearrow \widehat{f}$ has been evaluated) searching for sockets named y. Each time this effect is observed, the cost of the α-conversion scheme will be skewed by a factor proportional to the size of the subgraph R when no optimization is used, and by a constant factor when the (optimized) garbage collection rules are used.

One way of avoiding this is to ensure the reduction strategy prioritises reduction of the cuts which perform α-conversions. How these extra cuts are identified from renaming cuts that belong to the original circuit is also a point that needs to be addressed. In Example 6.1, this would involve evaluating the renaming cut $\widehat{\beta} \nearrow \widehat{f}$ in step 2, before the other cuts in the expression. This action would rightly prevent the cut $\widehat{\delta} \dagger \widehat{y}$ activating, then propagating, to the right.

Although the $\mathcal{X}$-calculus specifies three kinds of cut (inactive, left-activated and right-activated), an input term should only contain instances of inactive cuts[5]. When an inactive cut is chosen as the next redex, there is an implicit understanding that the cut should be run to 'completion'. That is, when we choose to execute a redex $P\widehat{\alpha} \dagger \widehat{x}Q$, we are making the choice to connect *all* α's in P with *all* x's in Q; a good reduction strategy will evaluate all cuts in this way. The following example enforces this view and illustrates why a naive outermost reduction strategy is not a favourable strategy in the setting of the $\mathcal{X}$-calculus.

[5]The system of active cuts was originally designed by Urban [24] to obtain a strongly normalising cut-elimination transformation on sequent proofs. They were specified as annotations on cut formulae used to direct cut-rule permutations to the leaves of a sequent derivation.

Example 6.2 We begin with a nesting of cuts between arbitrary terms P, Q, R, and S. A call-by-value left-most outermost reduction strategy is applied to the graph root node, which traverses the term attempting to match each rule with the current graph node. After a successful rewrite, the redex searching process restarts from the root node of the term graph. Recall that active cuts cannot propagate over each other.

$$\begin{array}{lll}
1. & (P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\beta} \dagger \widehat{y}R & \rightarrow (a\not\leftarrow) \\
2. & (P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\beta} \not\leftarrow \widehat{y}R & \rightarrow (cut\not\leftarrow) \\
3. & (P\widehat{\beta} \not\leftarrow \widehat{y}R)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\beta} \not\leftarrow \widehat{y}R) & \rightarrow (a\not\leftarrow) \\
4. & (P\widehat{\beta} \not\leftarrow \widehat{y}R)\widehat{\alpha} \not\leftarrow \widehat{x}(Q\widehat{\beta} \not\leftarrow \widehat{y}R) & \rightarrow \cdots
\end{array}$$

In this relatively simple example, we are left in a situation resembling a traffic-jam. By step (4), the propagation of the outermost cut is blocked by the innermost active cuts. When the innermost cut propagates down the graph one level, the second innermost cut is permitted to propagate down one level. This pattern expands to more complicated examples, where each outer cut follows in the wake of an innermost cut (as would be seen in a traffic-jam, one *cut* moves along a place, and each following *cut* shifts along, filling the empty space).

The overall effect of this is an undesired increase in the cost of searching for the next redex (which involves graph traversal, structural matching and checking side-conditions).

When a term graph is rewritten using the standard notion of graph rewriting, new nodes may be added to the graph. In step 1 of Example 6.2, an inactive cut is activated. Although the graph nodes Cut and CutL (for the cuts $\widehat{\delta} \dagger \widehat{z}$ and $\widehat{\delta} \not\leftarrow \widehat{z}$) are represented by two distinct node objects, our strategy must recognise that they are related in order to evaluate the cut to completion. The strategy will need to sequentially apply a number of rules to the term graph while 'following' the propagating cut through the expression.

Visser [26] proposes a generic language for specifying reduction strategies on term graphs. This language is rich enough to describe the kind of strategies we seek. The following section introduces the idea of *strategy combinators* and explains how to implement a reduction strategy for $\mathcal{X}$ that can reduce a cut to completion.

6.1 Strategy Combinators

Strategy combinators as defined by Visser [26, 11, 27] allow for the controlled traversal of a data structure. These combinators are then extended with rewrite rules, which allow on-site modification of the graph. Examples of some strategies that may be constructed with these combinators are: (1) normalize the graph using a set of rules, R, according to an innermost strategy; (2) repeatedly apply a rewrite rule to a node until failure; or (3) visit all nodes at level three of the graph, and so on.

For our purposes, it is sufficient to restrict ourselves to a subset of the language made up of the following combinators.

Definition 6.3 (STRATEGY COMBINATOR LANGUAGE [26])

s ::=	id	Identity
	| fail	Fail
	| L→R	Rewrite rule
	| [L→R]	List of Rewrite rules
	| seq(s,s)	Sequential Composition
	| choice(s,s)	Left-biased choice
	| all(s)	All immediate children
	| one(s)	One immediate child

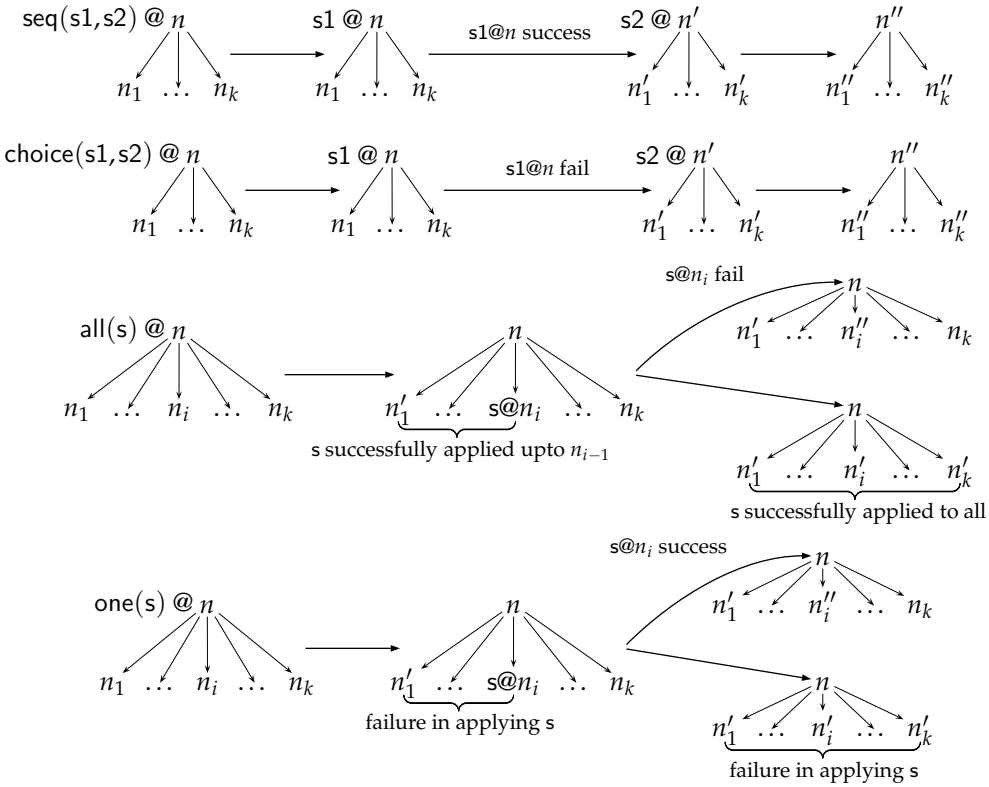


Figure 7: Applications of Basic Strategy Combinators to Arbitrary Graphs

Definition 6.4 (APPLICATION OF A STRATEGY) The application of a strategy to an $\mathcal{X}$-graph is a pair consisting of a strategy combinator, s, and a rooted subgraph (g|n). The strategy combinator system has a global *fail flag* which, when raised, indicates a *fail state*; this fail state affects the operational behaviour of some of the combinators.

We will write s@(n) for the application of the strategy combinator s to the $\mathcal{X}$ graph rooted at n.

The main results for the application of strategy combinators are either: (1) the strategy results in another set of strategies being applied to some node(s) of the graph; (2) the graph is modified by a rewrite rule; or (3) the state of the global fail flag is altered.

Figure 7 accompanies the following description of strategy combinators.

id@(n) : the identity strategy which simply leaves the supplied node unmodified.

fail@(n) : raise the fail flag indicating a state of failure.

L→R@(n) : attempt to match the left-hand side L with the subgraph rooted at n. If the match is successful, n will be rewritten to some subgraph rooted at n′ as dictated by the rewrite rule; any further strategies to be applied to n are updated to refer to n′. If the match is unsuccessful, the fail flag is raised.

[L→R]@(n) : sequentially traverse a list of rewrite rules while attempting to apply each rewrite rule to n. The strategy terminates the traversal of the list upon the successful application of a rewrite rule. If the list is exhausted and no rule was applicable, the fail flag is raised.

seq(s1,s2)@(n) : sequentially apply the argument strategies to n; if at any time a state of failure is reported the entire sequence strategy fails.

choice(s1,s2)@(n) : applies s1@(n), then applies s2@(n) if and only if s1@(n) reported a failure.

all(s)@(n) : attempt to apply s to each immediate successor (left-to-right) of the node n. Any

successive applications are aborted if at any point the fail flag is raised by the application of s to the immediate successors of n.

$\mathsf{one}(\mathsf{s})@(\mathsf{n})$: attempt to apply s left-to-right to a single immediate successor of n; if no successful application is found, the entire strategy fails.

In addition to these basic combinators, we will allow users to define their own complex combinators via the following construction.

$$\mathsf{C}(\mathsf{x}_1,\ldots,\mathsf{x}_k) \quad = \quad \mathsf{s}$$

C's arguments $\mathsf{x}_1,\ldots,\mathsf{x}_n$ may occur, and are bound, in the definition body, s. Definition 6.3 is then extended with a case for a user-defined combinator:

$$\begin{array}{lll} \mathsf{s} ::= & \ldots & \\ & \mid \mathsf{C}(\mathsf{s}_1,\ldots,\mathsf{s}_k) & \text{User-defined combinator} \end{array}$$

An application of a user-defined combinator to a node n, $\mathsf{C}(\mathsf{s}_1,\ldots,\mathsf{s}_k)@(\mathsf{n})$, denotes the instantiation $\mathsf{s}[\mathsf{x}_1 = \mathsf{s}_1,\ldots,\mathsf{x}_k = \mathsf{s}_k]$ of the body of s in the definition of C. Because this extension allows recursive strategies to be defined, we will dismiss nonsense definitions such as $\mathsf{C}(\mathsf{x}) = \mathsf{C}(\mathsf{x})$, by forbidding left-recursion.

We will also make use of some helper strategies, which can each be defined using the strategy language described above.

Definition 6.5 (HELPER STRATEGY COMBINATORS [27]) Below, we define some helper combinators, followed by an informal description of the effect of applying the combinator to a node of some term graph. We assume that, before the application of each strategy, the fail flag is not raised.

$\mathsf{try}(\mathsf{s}) = \mathsf{choice}(\mathsf{s},\mathsf{id})$: attempt to apply the argument strategy s to the node. If s fails, clear the fail flag.

$\mathsf{repeat}(\mathsf{s}) = \mathsf{try}(\mathsf{seq}(\mathsf{s},\mathsf{repeat}(\mathsf{s})))$: repeatedly apply the strategy s to the node, until no more applications are possible, leaving the system in an unfail state.

$\mathsf{oncetd}(\mathsf{s}) = \mathsf{choice}(\mathsf{s},\mathsf{one}(\mathsf{oncetd}(\mathsf{s})))$: search once top-down from the node and terminate after the first successful application of s. Raise the fail flag if no application was successful.

$\mathsf{outermost}(\mathsf{s}) = \mathsf{repeat}(\mathsf{oncetd}(\mathsf{s}))$: search depth-first from the node and attempt to apply s to each node of the term graph; after a successful application restart the search from the node on which the strategy was first called (modulo rewriting of the subgraph rooted at that node).

6.2 Reduction Strategies for $\mathcal{X}$

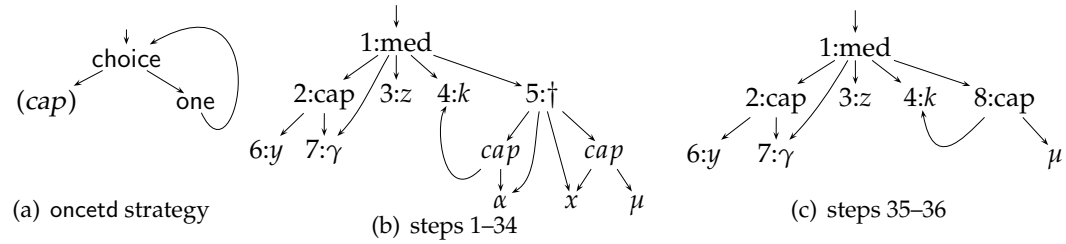
In this section we will define a strategy combinator that when applied to $\mathcal{X}$-graphs will evaluate an inactive cut to completion. We will then extend this strategy to work with our proposed solutions to the problems of name clash and capture.

First, however, we will give a detailed example of the steps involved in applying a rewrite rule strategy to an $\mathcal{X}$-graph following a simple traversal scheme. The example is intended to mimic the steps taken by our implementation.

Example 6.6 (A REDUCTION USING oncetd) In Figure 8, we illustrate the steps taken by the strategy language to apply $\mathsf{oncetd}(\mathit{cap})$ to the term graph

$$[\![\langle y\cdot\gamma\rangle\,\widehat{\gamma}\;[z]\;\widehat{k}(\langle k\cdot\alpha\rangle\,\widehat{\alpha}\;\dagger\;\widehat{x}\langle x\cdot\mu\rangle)]\!]_{\mathrm{G}}$$

During the application of a strategy, we maintain:



(a) oncetd strategy (b) steps 1–34 (c) steps 35–36

Figure 8: Application of oncetd (*cap*) to $\langle y{\cdot}\gamma\rangle\,\widehat{\gamma}\,[z]\,\widehat{k}(\langle k{\cdot}\alpha\rangle\,\widehat{\alpha}\,\dagger\,\widehat{x}\langle x{\cdot}\mu\rangle)$.

a node stack : a stack of node ids that records the path of the strategy through the term;

a combinator stack : a stack of combinator states which records the progress of the strategy; and

a failure flag : to record whether a strategy has resulted in failure.

The stack trace in Figure 8(a) begins with the root node of the term graph (Figure 8(b)) and the root node of the strategy graph (Figure 9) on the node stack and combinator stack, respectively; the failure flag is cleared. We write ch/i to indicate the choice strategy is currently evaluating its ith argument strategy, and one/j to indicate the one strategy currently is applying its argument strategy to the jth child of the node at which it was first applied. We summarise the interesting steps of the strategy below.

1-2 : The zeroth argument of choice is pushed onto the combinator stack.

3-4 : The failed match of the rule (*cap*) with node 1, results in a failure state. The choice combinator recovers from the failure, expanding its second argument.

5 : The one strategy pushes the zeroth child of node 1 (i.e. node 2) onto the node stack, and pushes its argument strategy (one) onto the combinator stack. Notice that one's argument strategy results in a recursive call being made to choice.

6-9 : The application of the rule (*cap*) to node 2 fails. The choice combinator recovers from the failure, pushing the one strategy onto the combinator stack. Since one is a traversal combinator, it pushes node 2's zeroth child onto the node stack.

10-14 : The application of the rule (*cap*) to node 6 fails, and the choice strategy once again recovers from the failure. However, notice that this time the one strategy also fails, since node 6 has no children.

15 : The combinators are popped off the stack until a combinator is found that can reset the failure flag. The combinator happens to be the one combinator which was evaluating the zeroth child of node 2. one clears the failure flag and proceeds to apply its argument strategy to the first child of node 2 (i.e. node 7).

16-19 : A repeat of the steps 10-14 occurs, except with node 7 on the node stack.

20-21 : Upon returning to the one combinator which was visiting the first child of node 2, it finds node 2 has no more children. Therefore, the one strategy fails and propagates the failure state.

22 : The one strategy failed to apply its argument strategy to the zeroth child of node 1 (i.e. node 2). It recovers from the failure state and attempts to apply the same argument strategy to the first child of node 1 (i.e. node 3).

23-34 : This application fails, and one applies the argument strategy to the second child of node 1 (i.e. node 4), then to the third child (node 5) when this fails.

35 : The application of the rule (*cap*) is successful and node 5 is rewritten to node 8 (see Figure

	Node Stack	Combinator Stack	Fail
1	1	-	
2	1	ch/0, (*cap*)	
3	1	ch/0	×
4	1	ch/1	
5	1,2	ch/1, one/0	
6	1,2	ch/1, one/0, ch/0, (*cap*)	
7	1,2	ch/1, one/0, ch/0	×
8	1,2	ch/1, one/0, ch/1	
9	1,2,6	ch/1, one/0, ch/1, one/0	
10	1,2,6	ch/1, one/0, ch/1, one/0, ch/0, (*cap*)	
11	1,2,6	ch/1, one/0, ch/1, one/0, ch/0	×
12	1,2,6	ch/1, one/0, ch/1, one/0, ch/1	
13	1,2,6	ch/1, one/0, ch/1, one/0, ch/1, one/0	
14	1,2,6	ch/1, one/0, ch/1, one/0, ch/1	×
15	1,2,7	ch/1, one/0, ch/1, one/1	
16	1,2,7	ch/1, one/0, ch/1, one/1, ch/0, (*cap*)	
17	1,2,7	ch/1, one/0, ch/1, one/1, ch/0	×
18	1,2,7	ch/1, one/0, ch/1, one/1, ch/1	
19	1,2,7	ch/1, one/0, ch/1, one/1, ch/1, one/0	
20	1,2,7	ch/1, one/0, ch/1, one/1, ch/1	×
21	1,2,7	ch/1, one/0, ch/1, one/1	×
22	1,3	ch/1, one/1	
23	1,3	ch/1, one/1, ch/0, (*cap*)	
24	1,3	ch/1, one/1, ch/0	×
25	1,3	ch/1, one/1, ch/1	
26	1,3	ch/1, one/1, ch/1, one/0	
27	1,3	ch/1, one/1, ch/1	×
28	1,4	ch/1, one/2	
29	1,4	ch/1, one/2, ch/0, (*cap*)	
30	1,4	ch/1, one/2, ch/0	×
31	1,4	ch/1, one/2, ch/1	
32	1,4	ch/1, one/2, ch/1, one/0	
33	1,4	ch/1, one/2, ch/1	×
34	1,5	ch/1, one/3	
35	1,5	ch/1, one/3, ch/0, (*cap*)	
36	1,8	ch/1, one/3, ch/0	
37	1	-	

Figure 9: Stack Traces

8(c)), and the node stack is updated.

36-37 : choice does not evaluate its second argument strategy since the failure state is clear when it is the head of the combinator stack. The remaining combinators are popped off the stack and the reduction terminates.

Many optimisations have been made to this reduction engine that can bypass a significant number of stack operations. For example, noticing that the Cap-nodes have only child nodes which are names (i.e. no variable nodes), steps 9-21 may be skipped.

In the previous section, we discussed some features we would require in an $\mathcal{X}$-calculus reduction strategy—these are summarised below.

Definition 6.7 (CRITERIA FOR EVALUATING A CUT) Given a pure $\mathcal{X}$-term, for any single *inactive* cut, a 'good' $\mathcal{X}$-calculus reduction strategy will:

i) Evaluate the inactive cut, $P\widehat{\alpha} \dagger \widehat{x}Q$, to completion so that all the plugs α in P are connected with all the sockets x in Q. The resultant term should have eliminated the cut $\widehat{\alpha} \dagger \widehat{x}$ from the term, as well as the connectors α and x.

ii) Transform a pure term to a pure term. This will involve 'following' related active cuts through the term until they are all destroyed.

iii) Give priority to α-conversion/renaming cuts over other cuts so that they do not interfere with parts ((a) and ((b).

We will consider the (call-by-name) $\mathcal{X}$-calculus rules as described in Section 2 to explain how the criteria is met. Realising that the $\mathcal{X}$-calculus rewrite rules are fixed, we can use our knowledge of the rules to guide the propagation of active cuts through $\mathcal{X}$-terms. We first define the following strategies as strict partitions over the set of rewrite rules (as ordered lists).

Definition 6.8

$$
\begin{array}{lcl}
\mathsf{rename} & = & [(cap),(exp),(med)] \\
\mathsf{logical} & = & [(exp\text{-}imp_{\mathrm{N}}),(exp\text{-}imp_{\mathrm{V}}),\mathsf{renaming}] \\
\mathsf{activate} & = & [(\not\backslash a),(a\not/)] \\
\mathsf{prop\_a1} & = & [(exp\text{-}ins\not/),(imp\not/),(cut\not/),(\not\backslash exp),(\not\backslash imp\text{-}ins),(\not\backslash cut)] \\
\mathsf{prop\_a2i0} & = & [(exp\text{-}outs\not/),(\not\backslash imp\text{-}outs)] \\
\mathsf{gc} & = & [(cap\not/),(\not\backslash cap)] \\
\mathsf{deact} & = & [(d\not/),(\not\backslash d)]
\end{array}
$$

Notice that, in fact, rule $(exp\text{-}imp_{\mathrm{V}})$ is obsolete here, since it will never match if $(exp\text{-}imp_{\mathrm{N}})$ does not.

Using the above combinators, we will describe a user-defined strategy combinator for reducing an inactive cut so that the criteria outlined in Definition 6.7 are obeyed.

The first two rule definitions (rename and logical) deal directly with inactive cuts. Their application is straightforward since they destroy the inactive cut being reduced. The next rule definition (activate) replaces an inactive cut with an active cut. To uphold part (c) of our criteria, we must continue evaluation of this active cut until it is destroyed. To that purpose, we will build a user defined strategy, propagate(), which propagates an active cut through a term until it is destroyed. Clearly propagate() will make use of the remaining definitions since they cover the cases that deal with activated cuts. We can now define a strategy evalcut() which evaluates a cut to completion.

```
evalcut() = choice(logical,
                   seq(activate, propagate())
            )
```

This strategy considers the *only* two cases of how to evaluate an inactivate cut: it can either be reduced by a logical rule, or activated and then propagated through the term. The remainder of this section looks at how to define the propagate() combinator.

We can break down the work that needs to be done by the propagate() combinator into four cases: (1) propagate an active cut through a circuit that *does not* mention any connectors involved in the cut; (2) propagate an active cut through a circuit that *does* mention the connectors involved in the cut; (3) garbage collect the active cut, since it has reached the level of the capsules; and (4) deactivate the cut, and attempt to reduce the deactivated cut.

The first two cases are covered by the following combinators.

```
(1) seq(prop_a1,
        all(try(propagate()))
    )
(2) seq(prop_a2i0,
        seq(all(all(try(propagate()))),
            evalcut()
    )   )
```

We explain these terms in the following paragraphs. First we remark that by nesting several 'all' combinators, we can visit all the nodes at a particular depth of a term graph (relative to the node at which the strategy was first applied). For instance, all(s)@(n) will apply s to the nodes at depth 1 relative to the node n, while all(all(s))@(n) will apply s to the nodes of n at

depth 2, and so on.

For (1), all the rules in the list prop_a1 have active cuts (that must be further propagated) at depth 1. Therefore, after a rule in prop_a1 has been applied, the propagate() strategy is applied to the nodes at depth 1 to propagate these newly introduced cuts.

For (2), all the rules of prop_a2i0 have active cuts at depth 2 *in addition to* an inactive cut at depth 0. The newly introduced active cuts at depth 2 must be further propagated, so there is a double nesting of the all combinator. The inactive cut at depth 0 also needs to be evaluated, so the evalcut() strategy is recursively applied; notice that to avoid the situation shown in Example 6.2, the active cuts are propagated *before* the inactive cut is evaluated.

The final two cases for the propagate strategy are described by the following combinators:

(3) gc
(4) seq(deact, evalcut())

For (3), the strategy gc is applicable if the active cut is with a capsule that does not introduce the connector bound by the active cut. The rewrite rules state that in this case the active cut should be destroyed. Therefore, following a successful application of a gc strategy, no further work needs to be done: there is no more active cut to propagate.

For (4), if the strategy deact is applicable, the active cut is deactivated, creating a new inactive cut at that level which must be evaluated; therefore the evalcut strategy is recursively applied to that inactive cut.

Combining these four parts, we obtain a definition for the propagate() strategy.

```
propagate() = choice(gc,
                     choice(seq(deact, evalcut()),
                            choice(seq(prop_a1,
                                       all(try(propagate()))),
                            ),
                            seq(    prop_a2i0,
                                    seq(all(all(try(propagate()))),
                                        evalcut()
                  )       )       )       )
```

The nesting of 'choice' combinators ensures each propagation case is considered at the current node. If no case is successful, the strategy combinator leaves the system in the fail state.

The above discussion shows how to evaluate a cut to completion in the CBN $\mathcal{X}$-calculus - notice that in activate, rule $(\curlywedge a)$ comes first, as does $(exp\text{-}imp_{\mathrm{N}})$ (see also Definition 6.9). We would like to extend this strategy to the sets of rules that solve the name clash and name capture problems described in Section 5. This involves working with a larger set of rules, but the same idea of evaluating a cut exists. We have two sets of rules to consider: the renaming using activated-cuts schemes and the lazy-copying α-conversion schemes with special rebinding symbols.

6.3 α-conversion with Rebinding Nodes

In this section, we look to define a 'good' strategy for evaluating an inactive cut for the solution of lazy copying as defined in Section 5.1. We can follow a similar strategy to that of the previous section, except some extra care must be taken with regards to the rebinding nodes. The nodes which perform the rebinding (rs and rp) are not part of the $\mathcal{X}$-calculus and would ideally be transparent.

Barendsen and Smetsers remark in [8]:

> *" mixing copy rules with the reduction rules [in the set of rewrite rules] may destroy properties [of the set of rewrite rules] such as confluence, or at least make it very difficult to check whether known properties [of the term graph rewriting system] extend to [the term graph rewriting system with copy rules added]. "*

This turns out to be true in our case; the rebinding nodes interfere with structural matching and the 'introduces' side conditions of the rewrite rules. For example, the term $\mathsf{rs}(\langle x{\cdot}\alpha\rangle, \alpha, \beta)$ does not introduce β, although it evaluates to $\langle x{\cdot}\beta\rangle$.[6] A naive solution could force the copying operation to completion, though this may turn out to be unnecessary.

We propose a lazier solution that *hides* the existence of rebinding nodes from the structural matching step of the rewriting procedure. Observing the lazy copying rewrite rules (Definition 5.2), this can be achieved by ensuring all rebinding nodes are at least two levels from any redex; in other words, a rebinding node should never be an immediate successor of a cut.

We specify a strategy pushRebind(), which when applied to a rebinding node, pushes that rebinding node down through the term-graph by one level. There are two cases to consider—either a rebinding rule (from Definition 5.1) is directly applicable to the current rebinding node, or not — i.e. there is a chain of one or more successive rebinding nodes prohibiting propagation. When such a chain exists, the strategy traverses to the lowest rebind node of that chain where a rule *will* be applicable. The lowest node is propagated one level further, followed in turn by each blocked ancestor.

The skeleton definition of this strategy is given below and makes use a group of all the rebinding evaluation rules (rebind_rules) as given in Definition 5.1, and a strategy repeat′(s) which repeatedly applies its argument strategy to the current node or fails.

```
repeat′()      = seq(s, repeat′(s))
pushRebind()   = choice(repeat′(rebind_rules),
                        choice(seq(all(try(pushRebind()))),
                                   rebind_rules
                               ),
                               fail
                 )       )
```

The propagate() strategy can now be extended to make use of pushRebind(). For each rebinding node introduced by a rewrite rule of the $\mathcal{X}$-calculus, the pushRebind() strategy is applied to that node, guaranteeing it is never the successor of a cut.

6.4 α-conversion with Renaming Cuts

The strategy evalcut() is first extended to cater for the case of $(\textit{exp-imp}_{\mathrm{N}})$ when a plug needs renaming. In this case, the active cut (at depth 2) is propagated through the term, using

```
seq(exp-imp_N,
    all(all(try(propagate()))))
)
```

[6] We could alter our definition of *introduces* so that $\mathsf{rs}(\langle x{\cdot}\alpha\rangle, \alpha, \beta)$ introduces β, but this will still not side-step the problem of the rebinding node structure interfering with the graph matching process.

The propagate() strategy should then be modified to cater for any additional α-conversion structure. This involves partitioning this larger set of rules into lists of rules which have cuts (to be further propagated) at common depths. The renaming cuts, which will be the innermost active cuts, must then be given priority over the other cuts in the rule. We will consider the variant of the rule $(\curlywedge exp)$:

$$Q\widehat{\alpha}\curlywedge\widehat{x}(\widehat{y}P\widehat{\beta}\cdot\delta) \;\rightarrow\; \widehat{w}(Q\widehat{\alpha}\curlywedge\widehat{x}(\langle w\cdot\mu\rangle\,\widehat{\mu}\curlywedge\widehat{y}P))\,\widehat{\beta}\cdot\delta \;\; (y\in fs(Q), \beta\notin fp(Q))$$

(the side-conditions of the rule do not come into play in the following discussion).

This rule would have been placed in a 'list of rewrite rules' combinator named prop_a1a2, indicating the rule has active cuts at a depth of 1 and also at a depth of 2; in the rule above these are respectively the cuts $\widehat{\alpha}\curlywedge\widehat{x}$ and $\widehat{\mu}\curlywedge\widehat{y}$ on right-hand side of the rule. The propagation of active cuts for this combinator is described by the combinator:

```
seq ( prop_a1a2,
    seq ( seq( all ( all ( try ( propagate () ) ) ),
               all ( try ( propagate () ) )
)   )   )
```

In other words, if a rule from the combinator prop_a1a2 is applied, propagate the inner active cuts at depth 2, and then propagate the remaining inner active cuts at depth 1. Lists of rules grouped together by the common depth of active and inactive cuts also need to be modified appropriately.

Now a 'good' outermost reduction strategy for the $\mathcal{X}$-calculus can be defined as:

```
outermost( evalcut() )
```

This strategy will search for the outermost inactive cut, which, when found, will apply the evalcut() strategy to that cut, evaluating that cut to completion.

6.5 Optimisations

We highlight a simple optimisation to the outermost strategy that will greatly decrease the search time for the next redex. Currently, after a successful reduction of an inactive cut, the search for the next inactive cut restarts from the point at which the original call to the strategy was made, i.e. the root node of the term graph. For a general outermost strategy, this is a safe course of action, since an outermost redex may have been skipped while an inner redex is evaluated c.f. call-by-value reduction in the λ-calculus. The $\mathcal{X}$-terms we evaluate are pure terms, and according to our evalcut() reduction strategy inactive cuts are evaluated to completion. Since inactive cuts cannot block propagating active cuts, the depth the subsequent outermost cuts are therefore guaranteed to be at a depth lower or equal to the current node. Using this fact, we can define an outermost reduction strategy to continue redex-searching from the current node pointed to.

```
outermost'(s)  =  seq(repeat(s),
                      all(try(outermost'(s)))
                  )
```

By parametising evalcut with an ordered list of activation rules and a variant of the $(exp\text{-}imp)$ rule, we can define two outermost strategies for call-by-name and call-by-value as follows.

Definition 6.9 (CBN AND CBV REDUCTION STRATEGIES)

$$\begin{array}{lcl}
\mathsf{cbnact} & = & [(\searrow a),(a\swarrow)] \\
\mathsf{cbvact} & = & [(a\swarrow),(\searrow a)] \\
\mathsf{outermostCBN} & = & \mathsf{outermost'}(\mathsf{evalcut}(\mathsf{cbnact},(\mathit{exp\text{-}imp}_{\mathrm{N}}))) \\
\mathsf{outermostCBV} & = & \mathsf{outermost'}(\mathsf{evalcut}(\mathsf{cbvact},(\mathit{exp\text{-}imp}_{\mathrm{V}})))
\end{array}$$

6.6 Benchmarks

In this section we present our benchmarks comparing the costs of the solutions to name clash and capture we proposed above. In the previous section we described how to extended our implementation with 'strategy combinators' in order to define a fair reduction strategy that could be used to compare the proposed solutions. Incorporating the strategy combinator language into our tool presented us with some problems, which we summarise below.

Visser has provided the community with a Java implementation of the strategy combinator framework, called JJTraveler. The framework allows modular extensions to the combinator language, allowing one to add *user-defined combinators* to the system by inheritance. A full description of this framework can be found in [27].

Integration of the framework with our $\mathcal{X}$ implementation, also written in Java, was straightforward. Unfortunately, preliminary testing revealed the implementation was unable to traverse some of the larger term-graphs generated by our benchmarks (which can contain in excess of 300,000 nodes) resulting in stack-overflows. The reason for this was the heavy reliance on recursion due to the use of a modified Visitor design-pattern[7]. We chose to re-implement the framework taking an iterative approach instead. The set of strategy combinators (Definition 6.3) extended with 'user-defined combinators' allows recursive strategies to be built. These were implemented as cyclic graphs (following the implementation of JJTraveller).

To maintain the state of the strategy (i.e., to track how much of the strategy had been processed during an application), we used two stacks. The working details of these structures were exemplified in Example 6.6. Recall that the 'combinator stack' tracked the current position within the strategy combinator graph, and the 'node stack' tracked the node of term graph which the current combinator was being applied to. The approach allowed us to obtain an accurate measure of the cost of *traversing* the graph searching for redexes: it was a count of the number of (constant-time) stack operations, plus the number of node matchings, plus the cost of checking the side-conditions.

We chose to measure the running of (interpreted) λ-terms since these are well-known benchmarks [1, 19], and the efficiency of the various formalisms and abstract machines can be better compared. We can of course not confront their (published) run-time measurements because of differences in platforms and processor architectures.

We use the usual encoding for Church Numerals ($\mathsf{n} = \lambda xy.x^n y$). In addition, we use the combinators $P = (\lambda z.(\lambda x.zxxx)(\lambda y.\mathsf{2}(\lambda x.y(x\mathsf{I}))n))\mathsf{II}$, with n replaced by a chosen Church Numeral and $\mathsf{I} = \lambda x.x$.

Note that we do not wish to consider the cost of encoding λ-terms to $\mathcal{X}$ as we are interested in comparing the efficiency of the α-conversion mechanisms for the systems presented; because of this, we use Prawitz's encoding (Definition 2.19). Our benchmarking results are listed in Figures 10 and 11. For each test case we record the following two measurements:

[7]Simply increasing the stack size of the JVM was not seen to be a scalable solution. Although recent Java implementations do include recursion optimisations, these mainly work on performance. Since our benchmarks will count atomic operations rather than measure time, our main concern is heap usage, which we can manage more efficiently with an iterative approach.

TestCase	rebind - GC		α-safe - GC		α-safe + GC		avoid capt. - GC	
	Search	Rewrite	Search	Rewrite	Search	Rewrite	Search	Rewrite
22II	0.861	0.163	0.264	0.0765	0.159	0.0421	0.0994	0.0330
222II	62.0	15.7	3.64	0.942	0.841	0.208	0.495	0.153
2222II	-	-	-	-	4650	715	1690	522
210II	0.372	0.0682	0.140	0.0401	0.0762	0.0214	0.0548	0.0194
2210II	2.40	0.471	0.649	0.165	0.269	0.0660	0.173	0.0541
22210II	225	56.7	10.6	1.84	1.53	0.305	0.805	0.236
P2	32.9	7.85	4.01	0.857	0.999	0.218	0.429	0.145
P3	51.4	12.5	5.54	1.11	1.20	0.256	0.487	0.165
P5	110	27.2	9.94	1.74	1.63	0.331	0.602	0.206
P10	429	109	31.4	4.01	2.98	0.519	0.890	0.309
P20	2240	575	143	11.6	6.71	0.895	1.47	0.514
P50	26100	6780	1550	58.4	26.3	2.02	3.19	1.13

Figure 10: CBV Results: Cost measured in units of 10^6 operations (to 3.s.f)

TestCase	rebind + GC		α-safe - GC		α-safe - GC		avoid capt. - GC	
	Search	Rewrite	Search	Rewrite	Search	Rewrite	Search	Rewrite
22II	0.914	0.156	0.314	0.0908	0.173	0.0439	0.0963	0.0322
222II	17.0	3.27	3.86	1.01	1.42	0.305	0.551	0.178
2222II	-	-	146000	29700	13100	1460	2490	805
210II	0.458	0.0819	0.164	0.0441	0.0868	0.0227	0.0559	0.0198
2210II	2.12	0.360	0.726	0.177	0.315	0.0742	0.151	0.0507
22210II	32.1	6.38	7.29	1.60	2.11	0.441	0.754	0.246
P2	540	138	22.5	3.02	3.51	0.582	0.902	0.308
P3	875	224	30.6	3.67	4.23	0.670	1.04	0.354
P5	2000	515	52.6	5.22	5.85	0.847	1.32	0.447
P10	8670	2240	152	10.5	11.0	1.29	2.02	0.679
P20	49600	12900	624	27.2	25.9	2.17	3.41	1.14
P50	626000	163000	6050	126	107	4.81	7.59	2.54

Figure 11: CBN Results: Cost measured in units of 10^6 operations (to 3.s.f)

Search Cost : a count of atomic operations involved in traversing the graph and searching for redexes, i.e. the number of push/pop stack operations to evaluate the strategy, plus the number of attempted matchings made between the rule heads and graph nodes, plus the cost of testing the side-condition.

Rewrite Cost : a count of the (more expensive) graph transforming operations, i.e. the number of nodes added and deleted plus the number of edges added and deleted.

A straightforward numerical comparison of costs suggests the following relationship of efficiency between α-conversion schemes, under either reduction strategy (CBN or CBV).

$$\text{Rebinding GC} < \alpha\text{-Safe - GC} < \alpha \text{ + Safe GC} < \text{Capture Avoid GC}$$

Another observable trend is the linear relationship between the redex search cost and graph rewrite cost. As the size of the graph increases, the search cost increases since more operations are required to traverse the graph structure. Once a suitable redex (a cut) is found, the cost of reducing that cut is related to the size of its subterms since it must be propagated through them.

There is a significant difference between the cost of reduction under the rebinding scheme

versus the other schemes tested. In fact, our original attempt at implementing the rebinding solution was not lazy at all, and often resulted in memory requirements greater than the 2GiB limit. We recorded the size of the graph (number of node objects) as the reduction progressed for each test-case in order to gain insight into this vast requirement of system resources. Results from the eager strategy are shown in Figures 12(a) and 13(a). As explained in Section 5.1, the rebinding scheme works by destroying the sharing in portions of the term-graph as so to guarantee that no binder occurs more than once. This copying-out effect, seen as peaks in the graphs, shows an increase in the number of nodes whenever a cut is propagated through varying sizes of subterm causing all sharing to be destroyed. Looking closer at these graphs the cost of searching is also clearly visible. As the number of nodes in a graph increases, small horizontal 'platforms' can be observed. These regions represent pure search costs consisting only of traversal stack operations plus unsuccessful rule matches.

Switching to a lazy mechanism as detailed in Section 6.3, although still relatively expensive, kept the size of the graph low enough for many previously failed tests to run to completion. The results of the lazy strategy are shown in Figures 12(b) and 13(b). The CBV graph highlights nicely the copying out of each argument when it is supplied to a function.

We also investigated the variation of graph size under the α-conversion schemes; the results are displayed in Figures 12(c) and 13(c). The shapes of these graphs appear to be less random than those from the rebinding scheme, and we see that for a particular reduction strategy (CBN or CBV) the overall shape of the graphs are similar. This is an expected side effect of the reduction strategy $\mathsf{evalcut}()$, designed to make α-conversions transparent from the point of view of the term being reduced. The reductions therefore only diverges at points where the need for α-conversions differ.

Conclusions and future work

We have studied various solutions for the problem of name capture in the context of a term-graph rewriting implementation of the $\mathcal{X}$-calculus. The first uses a rebinding technique, that required an extension of the syntax, with additional rules. The second and third change the reduction rules of $\mathcal{X}$, but without extending the signature. The second solution guarantees that (generated) term adhere to Barendregt's convention, whereas the third checks for captivation of free names. We have measured the efficiency of all these solutions, and conclude that the latter, although syntactically very close to the second, is by far the best.

There are a number of questions still open that will be investigated in future work. First of all, we need to understand why the CBN measurement of the pure α-safe system are only for the Q terms so much worse than the CBV measurements. We aim to show that the reduction strategy used in the tool is normalising.

References

[1] A. Asperti, C. Giovanetti, and A. Naletto. The Bologna Optimal Higher-Order Machine. *Journal of Functional Programming*, 6(6):763–810, 1996.

[2] S. van Bakel, S. Lengrand, and P. Lescanne. The language $\mathcal{X}$: Circuits, Computations and Classical Logic. In M. Coppo, Elena Lodi, and G. Michele Pinna, editors, *Proceedings of Ninth Italian Conference on Theoretical Computer Science* (ICTCS'05), *Siena, Italy*, volume 3701 of *Lecture Notes in Computer Science*, pages 81–96. Springer Verlag, 2005.

[3] S. van Bakel and P. Lescanne. Computation with Classical Sequents. *Mathematical Structures in Computer Science*, 18:555–609, 2008.

[4] S. van Bakel and J. Raghunandan. Implementing $\mathcal{X}$. In *Electronic Proceedings of Second International Workshop on* Term Graph Rewriting 2004 *(TermGraph'04), Rome, Italy*, Electronic Notes in Theoretical Computer Science, 2005.

[5] H. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

[6] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe,* Eindhoven, The Netherlands, volume 259-II of *Lecture Notes in Computer Science*, pages 141–158. Springer Verlag, 1987.

[7] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Towards an Intermediate Language based on Graph Rewriting. In *Proceedings of PARLE, Parallel Architectures and Languages Europe,* Eindhoven, The Netherlands, volume 259-II of *Lecture Notes in Computer Science*, pages 159–175. Springer Verlag, 1987.

[8] E. Barendsen and S. Smetsers. Extending Graph Rewriting with Copying. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Proceedings of International Workshop 'Graph Transformations in Computer Science',* Daghstuhl, Germany, January 1993, volume 776 of *Lecture Notes in Computer Science*, pages 51–70. Springer Verlag, 1994.

[9] E. Barendsen and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. *Mathematical Structures in Computer Science*, 1996.

[10] R. Bloo and K.H. Rose. Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Substitution and Garbage Collection. In *CSN'95 – Computer Science in the Netherlands*, pages 62–72, 1995.

[11] M. Bravenboer, K. Trygve Kalleberg, R. Vermaas, and E. Visser. *Stratego/XT Tutorial, Examples, and Reference Manual for Stratego/XT 0.16*. Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, November 2005.

[12] A. Church. A Note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1(1):40–41, 1936.

[13] P.-L. Curien and H. Herbelin. The Duality of Computation. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, volume 35.9 of *ACM Sigplan Notices*, pages 233–243. ACM, 2000.

[14] G. Gentzen. Investigations into logical deduction. In *The Collected Papers of Gerhard Gentzen*. Ed M. E. Szabo, North Holland, 68ff (1969), 1935.

[15] Stefano Guerrini. Sharing Implementations of Graph Rewriting Systems. *Electronic Notes in Theoretical Computer Science*, 127(5):113–132, 2005.

[16] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J de Vries. The adequacy of term graph rewriting for simulating term rewriting. In M.R. Sleep, M.J. Plasmeijer, and M.C.D.J. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 157–168. J. Wiley & Sons, 1993.

[17] S.C. Kleene. *Introduction to Metamathematics*. North Holland, Amsterdam, 1952.

[18] S. Lengrand. Call-by-value, call-by-name, and strong normalization for the classical sequent calculus. In B. Gramlich and S. Lucas, editors, *3rd Workshop on Reduction Strategies in Rewriting and Programming (WRS 2003)*, volume 86 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[19] I. Mackie. Efficient lambda-Evaluation with Interaction Nets. In *RTA*, pages 155–169, 2004.

[20] M. Parigot. An algorithmic interpretation of classical natural deduction. In *Proceedings of 3rd International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'92)*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer Verlag, 1992.

[21] D. Prawitz. *Natural Deduction, A Proof-Theoretical Study*. Almqvist & Wiksell, 1965.

[22] R. Sleep, M.J. Plasmeijer, and M.C.J.C van Eekelen, editors. *Term Graph Rewriting. Theory and Practice*. Wiley, 1993.

[23] A.J. Summers. Interpretation of λ-calculus terms to $\mathcal{X}$ in as suggested by Prawitz. Personal communication, 2007".

[24] C. Urban. *Classical Logic and Computation*. PhD thesis, University of Cambridge, October 2000.

[25] C. Urban and G.M. Bierman. Strong normalisation of cut-elimination in classical logic. *Fundamenta Informaticae*, 45(1,2):123–155, 2001.

[26] E. Visser and Z. Benaissa. A core language for rewriting. *Electronic Notes in Theoretical Computer Science*, 15, 1998.

[27] J. Visser. Visitor Combination and Traversal Control. In *OOPSLA*, pages 270–282, 2001.

[28] P. Wadler. Call-by-Value is Dual to Call-by-Name. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 189 – 201, 2003.

[29] C.P. Wadsworth. *Semantics and pragmatics of the lambda calculus*. PhD thesis, Oxford University, 1971. Thesis CST-33-85.

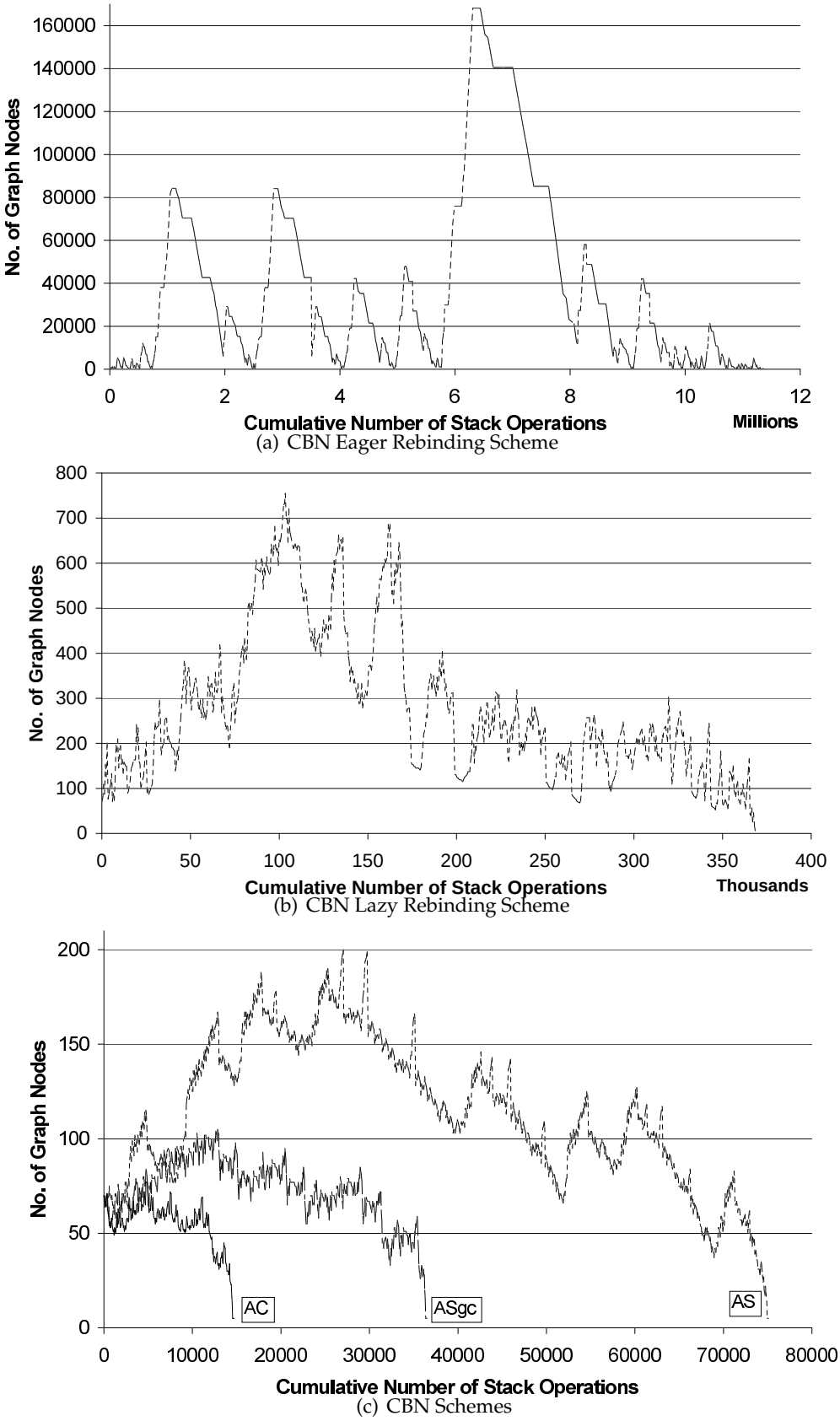


Figure 12: Variation of Graph Size over 'time' for reducing $\llbracket 222II \rrbracket_\delta$ under different α-conversion schemes (AS=αSafe-NoGC, ASgc=αSafe-GC, AC=AvoidCapture-GC)

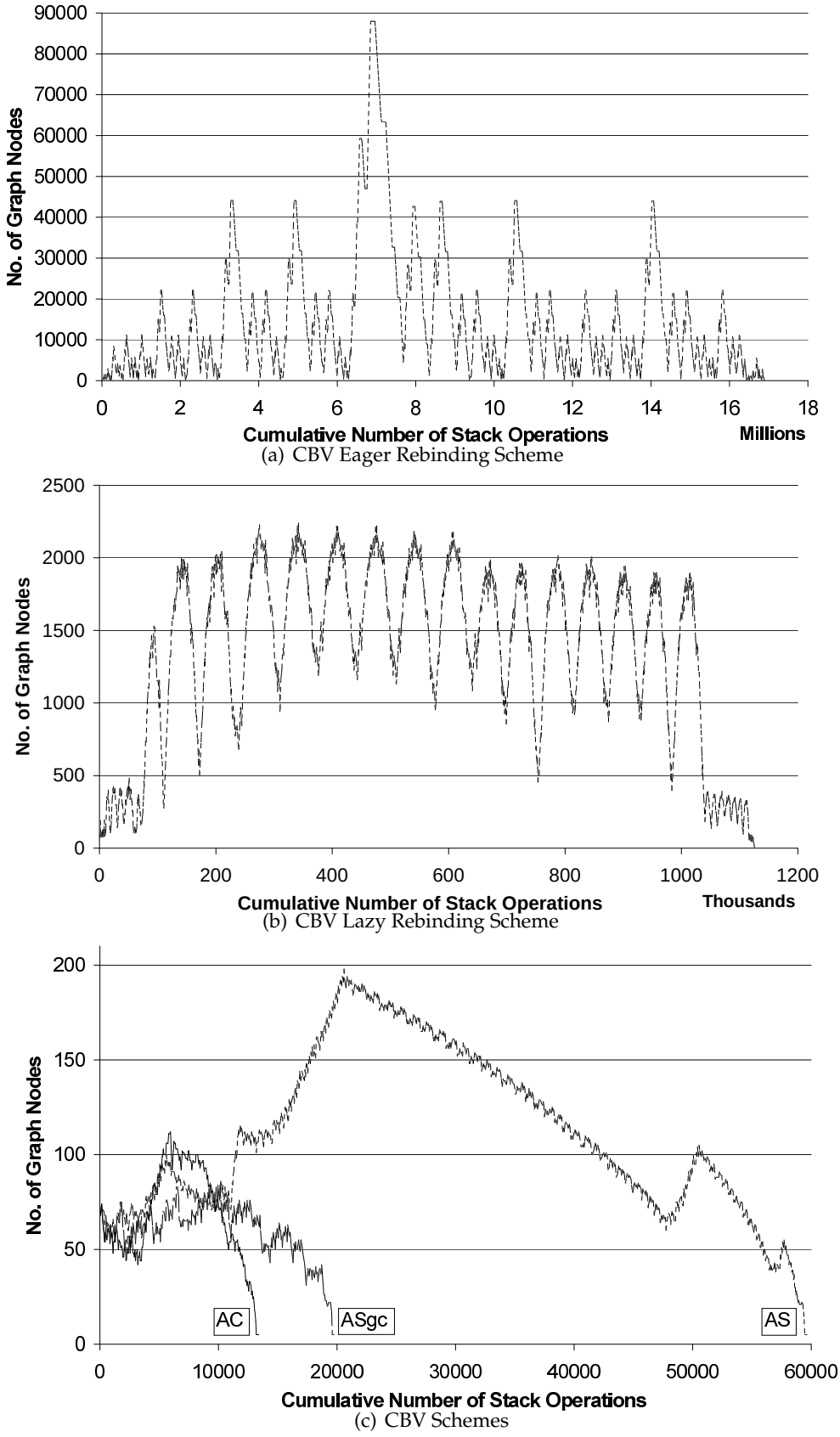


Figure 13: Variation of Graph Size over 'time' for reducing $[\![222II]\!]_\delta$ under different α-conversion schemes (AS=αSafe-NoGC, ASgc=αSafe-GC, AC=AvoidCapture-GC)