Explicit Alpha Conversion and Garbage Collection in $\mathcal{X}$

Extended Abstract

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

In this paper we study the calculus of circuits $\mathcal{X}$, as first presented in [13] and studied in detail in [2]. We will present improvements on the implementation for $\mathcal{X}$ using term graph rewriting techniques that was presented in [3], which result in a far more efficient running of the reduction engine. We show that alpha conversion can be dealt with 'on the fly', by implementing the avoidance of capture by modifying the rewrite rules. We then study two different approaches to garbage collection, and compare the various implementations by presenting benchmarks.

Introduction

This paper will present improvements on the term graph rewriting model of implementation for the (untyped) calculus $\mathcal{X}$ as presented in [3]. $\mathcal{X}$ is a new style calculus which embodies both substitution and context call, that has first been defined in [13] and was later extensively studied in [2].

The origin of $\mathcal{X}$ lies within the quest for a language designed to give a Curry-Howard correspondence to the sequent calculus for Classical Logic. Starting from different approaches in that area —we mention Herbelin and Curien [11] and Urban [16]— in [13] the calculus $\mathcal{X}$ was introduced, and studied in the context of the $\overline{\lambda}\mu\tilde{\mu}$-calculus [11].

We should point out that [13], as well as [16], did not study any property of *untyped* $\mathcal{X}$, but focused only on its typed aspects in connection with the sequent calculus. Urban [16] set out to study the structure of proofs, so the terms there carry types and correspond only to proofs. In the approach to $\mathcal{X}$ we take 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 normalisation, recursion, normalising strategies, confluence, etc.

While studying $\mathcal{X}$ as an untyped language, some unexpected special properties surfaced: it became apparent that $\mathcal{X}$ provides an excellent general purpose machine, very well suited to encode various calculi; in [2], the expressive power of $\mathcal{X}$ is illustrated there by giving consistent interpretations of calculi λ, $\lambda\mathbf{x}$, $\lambda\mu$, and $\overline{\lambda}\mu\tilde{\mu}$. Amongst the calculi studied in that paper, the Calculus of Explicit Substitutions, $\lambda\mathbf{x}$, stands out. Also, the calculus is actually symmetric [5]; the '*cut*', represented by $(P\widehat{\alpha} \dagger \widehat{x}Q)$ represents, in a sense, the explicit substitution of P for x in Q, but also that of Q for α in P.

Perhaps the main feature of $\mathcal{X}$ is that it constitutes a *variable* and *substitution*-free method of computation. Rather than having variables like x representing places where terms can be inserted, in $\mathcal{X}$ (where we speak of *circuits*) the symbol x represents a *socket*, to which a circuit

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 $\mathcal{X}$ shows nicely how the interaction between the two subtly and gently percolates through the circuits.

Although the origin of $\mathcal{X}$ is a logic, and one could expect it to be close to the λ-calculus, it is in fact specified as a *conditional term rewriting system*; the non-standard aspects are the presence of binding, and that the rewrite rules are defined using *three* different classes of open (variable) nodes (for plugs, sockets, and circuits).

This observation is in fact the reason for the research which led to the present paper. It was decided to build an interpreter for $\mathcal{X}$, so that researchers interested could familiarise themselves with the reduction engine and, more importantly, with the calculus. A tool[1] was developed using the term graph rewriting technology, that allows users to input circuits from $\mathcal{X}$. 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 and α-conversion. In [3] 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 show that the special character of $\mathcal{X}$, being a conditional term rewriting system, makes it possible to study α-conversion *on the level of the language itself*. This study is of course motivated by the implementation issues in the tool built; we apply existing concepts to new situations, finding innovative solutions. In $\mathcal{X}$, α-conversion is not just an implementation issue, but a first-class citizen (as is substitution). The expressiveness of $\mathcal{X}$ makes it an ideal implementation language, where no important details of computation are hidden. This allows us to directly measure the cost of α-conversion using the same currency as for the cost of substitution and redex-contraction. With minor changes to the rules we can test all different solutions on the level of the language itself.

This paper does not claim to find innovative solutions to the problem of α-conversion itself; many solutions to this problem exists, and we have, so far, chosen standard approaches. What we do achieve is 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 two solutions for the problem of α-conversion that are both expressed as changes to the rules: one will preserve Barendregt's convention, the other avoids the capture of free connectors by binders. Much to our surprise, the resulting rewrite rules of these two solutions are very similar, but for the fact that in the second freeness is an issue, rather than being bound as in the first. The reduction engine thus obtained proved to be impressively much more efficient, especially after the addition of two different notions of garbage collection.

[1] `http://www.doc.ic.ac.uk/~jr200/X`

$$
\begin{array}{rcl}
\llbracket x \rrbracket & = & \langle x \mid \emptyset \rangle \\
\llbracket \alpha \rrbracket & = & \langle \alpha \mid \emptyset \rangle \\
\llbracket \langle x \cdot \alpha \rangle \rrbracket & = & \langle r \mid \{r : \mathsf{cap}(r_1, r_2)\} \cup G_1 \cup G_2 \rangle, \\
 & & \text{where } \langle r_1 \mid G_1 \rangle = \llbracket x \rrbracket, \\
 & & \quad \langle r_2 \mid G_2 \rangle = \llbracket \alpha \rrbracket \\
\llbracket P \widehat{\alpha} \dagger \widehat{x} Q \rrbracket & = & \langle r \mid \{r : \mathsf{cut}(r_1, r_2, r_3, r_4)\} \\
 & & \cup G_1 \cup G_2 \cup G_3 \cup G_4 \rangle \\
 & & \text{where } \langle r_1 \mid G_1 \rangle = \llbracket P \rrbracket \\
 & & \quad \langle r_2 \mid G_2 \rangle = \llbracket \alpha \rrbracket \\
 & & \quad \langle r_3 \mid G_3 \rangle = \llbracket x \rrbracket \\
 & & \quad \langle r_4 \mid G_4 \rangle = \llbracket Q \rrbracket
\end{array}
$$

$$
\begin{array}{rcl}
\llbracket P \widehat{\alpha} [y] \widehat{x} Q \rrbracket & = & \langle r \mid \{r : \mathsf{med}(r_1, r_2, r_3, r_4, r_5)\} \\
 & & \cup G_1 \cup G_2 \cup G_3 \cup G_4 \cup G_5 \rangle \\
 & & \text{where } \langle r_1 \mid G_1 \rangle = \llbracket P \rrbracket \\
 & & \quad \langle r_2 \mid G_2 \rangle = \llbracket \alpha \rrbracket \\
 & & \quad \langle r_3 \mid G_3 \rangle = \llbracket y \rrbracket \\
 & & \quad \langle r_4 \mid G_4 \rangle = \llbracket x \rrbracket \\
 & & \quad \langle r_5 \mid G_5 \rangle = \llbracket Q \rrbracket \\
\llbracket \widehat{y} P \widehat{\alpha} \cdot \beta \rrbracket & = & \langle r \mid \{r : \mathsf{exp}(r_1, r_2, r_3, r_4)\} \\
 & & \cup G_1 \cup G_2 \cup G_3 \cup G_4 \rangle \\
 & & \text{where } \langle r_1 \mid G_1 \rangle = \llbracket \beta \rrbracket \\
 & & \quad \langle r_2 \mid G_2 \rangle = \llbracket y \rrbracket \\
 & & \quad \langle r_3 \mid G_3 \rangle = \llbracket P \rrbracket \\
 & & \quad \langle r_4 \mid G_4 \rangle = \llbracket \alpha \rrbracket
\end{array}
$$

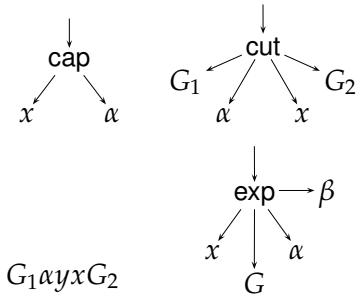
Term graph interpretation of circuits

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

Although the origin of $\mathcal{X}$ is a logic, and one could expect it to be close to the λ-calculus, it is in fact specified as a *conditional term rewriting system*; the non-standard aspects are a notion of binding, and that the rewrite rules are defined using *three* different classes of variable nodes (for plugs, sockets, and circuits). In our view, for a general term rewriting system the term-graph technology is the best platform for an efficient implementation: this prompted us to build an interpreter of $\mathcal{X}$ using this framework, as first reported on in [3]. The technique applied is the conventional one of [15, 7, 8], where terms and rewrite rules are *lifted* to graphs. We used 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 united via shared leaves. Term graphs are defined by:

Definition 1.1 (Term Graphs) Using the signature $\{\mathsf{cap}, \mathsf{cut}, \mathsf{cutl}, \mathsf{cutr}, \mathsf{exp}, \mathsf{med}\}$, an infinite set of *labels* ranged over by k, l, m, n and the set of connectors, we define (ordered) graphs by the following grammar:

$$
\begin{array}{rcl}
G & ::= & k : \mathsf{cap}\,(l{:}x, m{:}\alpha) \mid \\
 & & k : \mathsf{cut}\,(G_1, l{:}\alpha, m{:}x, G_2) \mid \\
 & & k : \mathsf{cutl}\,(G_1, l{:}\alpha, m{:}x, G_2) \mid \\
 & & k : \mathsf{cutr}\,(G_1, l{:}\alpha, m{:}x, G_2) \mid \\
 & & k : \mathsf{exp}\,(l{:}x, G, m{:}\alpha, n{:}\beta) \mid \\
 & & k : \mathsf{med}\,(G_1, l{:}\alpha, m{:}y, n{:}x, G_2)
\end{array}
$$

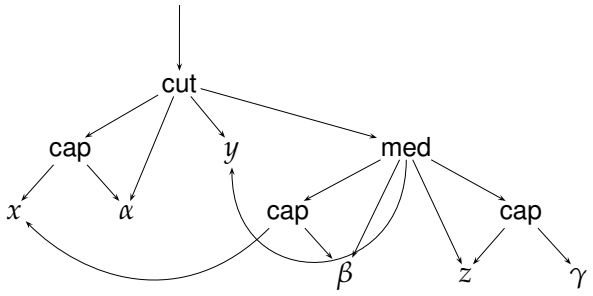


Notice that we do not need to use the $\widehat{\cdot}$, since it is immediately clear which connectors are binding occurrences. In the last graph, for example, the connectors x, α and β could appear in G; then edges would point out from G to these connectors. This leads to the following formal definition of interpreting circuits in $\mathcal{X}$ as graphs (using the notation of [9]).

Definition 1.2 (GRAPH INTERPRETATION) For each circuit P, its *graph interpretation*, $\mathbb{P}P\mathbb{J}$, expressed as a pair consisting of a label for the *root* of the graph and its *edges* is defined in Fig. .

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

Example 1.3 $\mathbb{P}\langle x{\cdot}\alpha\rangle\,\widehat{\alpha}\dagger\widehat{y}(\langle x{\cdot}\beta\rangle\,\widehat{\beta}\;[y]\;\widehat{z}\langle z{\cdot}\gamma\rangle)\mathbb{J}$ becomes



Definition 1.4 ($\mathcal{X}$-GRAPHS)
i) We define the set of *initial* $\mathcal{X}$-graphs as the image of $\mathcal{X}$ circuits under $\mathbb{P}\cdot\mathbb{J}$.

ii) The *lifting* of the reduction rules to term graph rewriting rules is expressed by:

$$\begin{array}{rcl}\mathbb{P}\mathit{left}\to\mathit{right}\mathbb{J} & = & \langle r_l \mid G_l \cup G_r\rangle \\ & & \text{where } \langle r_l \mid G_l\rangle \;=\; \mathbb{P}\mathit{left}\mathbb{J} \\ & & \quad\;\; \langle r_r \mid G_r\rangle \;=\; \mathbb{P}\mathit{right}\mathbb{J}\end{array}$$

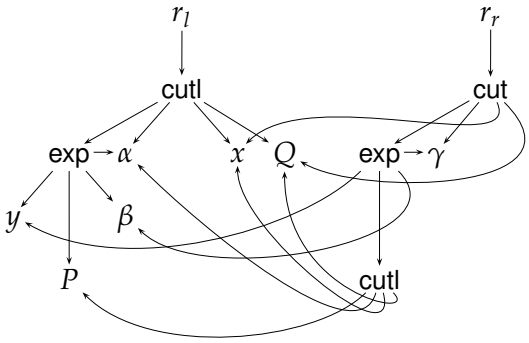
These rules induce a notion $G \to_{\mathrm{G}} G'$ of term graph rewriting.

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

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

Example 1.5 The result of lifting rule (*exp-out*$\not\leftarrow$) to a term graph rule becomes:

$$\begin{array}{rcl}\langle r_l \mid \{r_l & : & \mathsf{cutl}(1,2,3,4) \\ 1 & : & \mathsf{exp}(5,6,7,2) \\ 2 & : & \alpha \\ 3 & : & x \\ 4 & : & Q \\ 5 & : & y \\ 6 & : & P \\ 7 & : & \beta \\ r_r & : & \mathsf{cut}(8,9,3,4) \\ 8 & : & \mathsf{exp}(5,10,7,9) \\ 9 & : & \gamma \\ 10 & : & \mathsf{cutl}(6,2,3,4)\}\rangle\end{array}$$



Notice that an application of this rule would 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.

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

Definition 1.6 *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.

We have the following adequacy result for the term graph rewriting engine:

Theorem 1.7 *Let G_1, G_2 be $\mathcal{X}$-graphs, and P_1, P_2 be $\mathcal{X}$-circuits such that $Unrv\ G_i = \llbracket P_i \rrbracket$, 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: Straightforward.

We can now prove the following result:

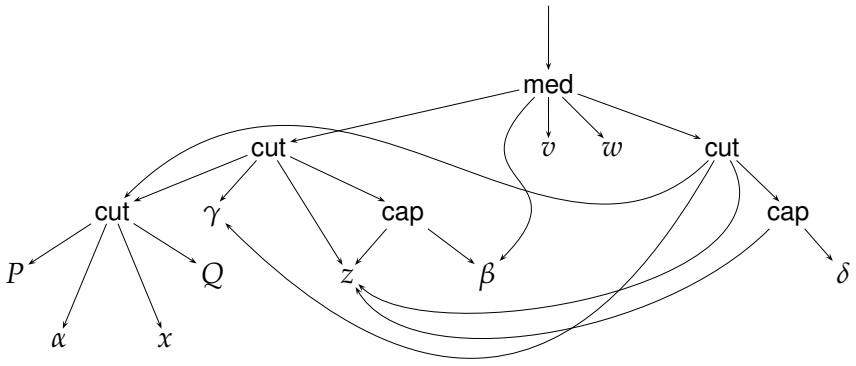
Theorem 1.8 *If $P \to Q$ in one step, then there exists a $\mathcal{X}$-graph G such that: $\llbracket P \rrbracket \to_{\mathrm{G}} G$, and $Unrv\ G = \llbracket Q \rrbracket$.*
Proof: Easy.

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

Example 1.9 Let P and Q be pure 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 $z \neq v$):

$$\begin{array}{l} (P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\gamma} \nwarrow \widehat{z}(\langle z{\cdot}\beta\rangle \widehat{\beta}\ [v]\ \widehat{w}\langle z{\cdot}\delta\rangle) \ \to\ (\nwarrow cut), (\nwarrow d), (\nwarrow d) \\ ((P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\beta\rangle)\widehat{\beta}\ [v]\ \widehat{w}((P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\delta\rangle) \\ \qquad\qquad \to\ (a\not\nwarrow), (gc\not\nwarrow), (\nwarrow a), (\nwarrow gc) \\ (P\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\beta\rangle)\widehat{\beta}\ [v]\ \widehat{w}(Q\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\delta\rangle) \end{array}$$

We cannot simulate this in our term-graph rewriting engine. Instead, we get



for the circuit in the second line. Of course, the cut $\mathsf{cut}(P, \alpha, x, Q)$ can be reduced only *once*, 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. We need to investigate the confluence of, for example, CBN and CBV-reduction strategies before we can strengthen the above result.

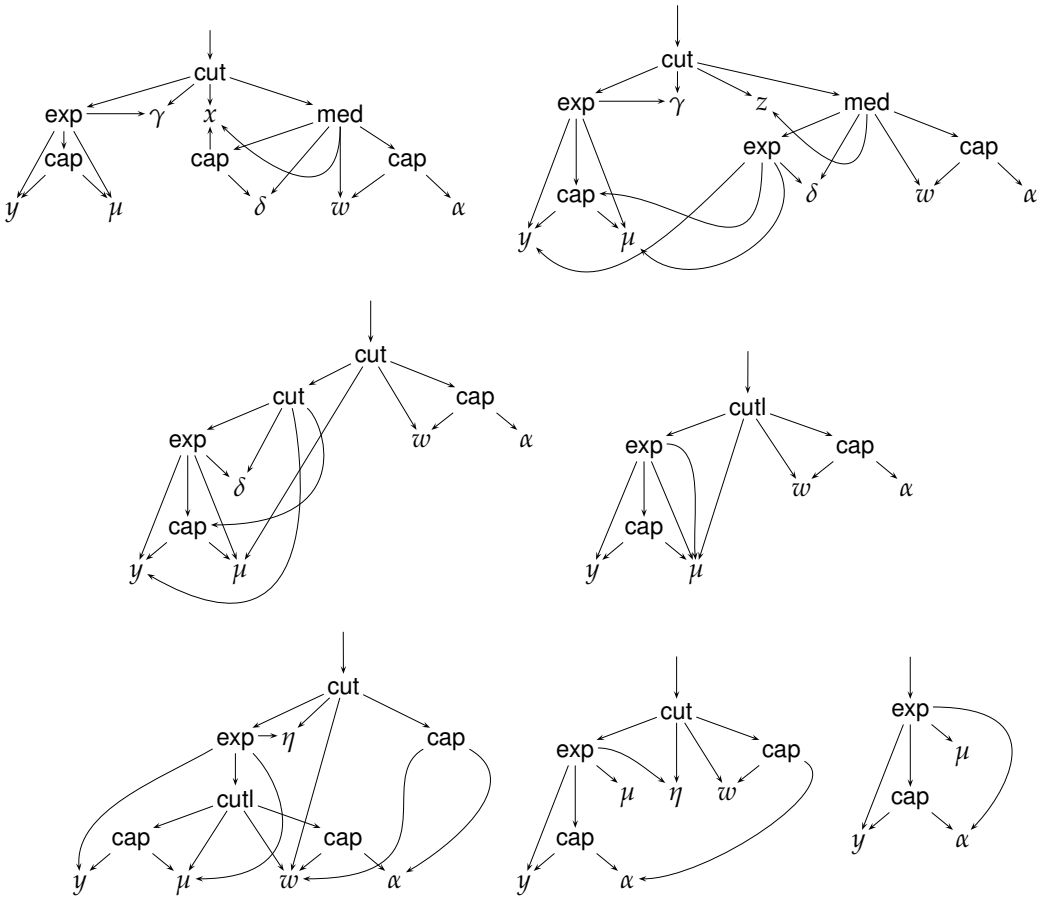


Figure 1: Alpha-conversion problem when reducing $[\![(\lambda x.xx)(\lambda y.y)]\!]^{\lambda}_{\alpha}$

2 Dealing with α-conversion

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

$$(\lambda xy.xy)(\lambda xy.xy) \;\rightarrow\; \lambda y.(\lambda xy.xy)y \;\rightarrow\; \lambda yy.yy$$

The conflict is caused by the fact that during the second reduction step, the free occurrence of y is brought *under* the binding.

A particular problem in dealing with α-conversion here is that the only reduction rule is $(\lambda x.M)N \rightarrow M[N/x]$, where the substitution is *implicit*, and supposed to be performed *immediately*. For example, when reducing $(\lambda xy.xy)(\lambda xy.xy) \rightarrow \lambda y.(\lambda xy.xy)y$, 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. This makes α-conversion difficult to tackle in the context of the pure λ-calculus.

To consider the substitution as a separate syntactic structure implies moving from the λ-calculus to $\lambda\mathbf{x}$ [10], and is of course a natural step when moving towards an implementation.

We will now present a solution for the α-conversion problem in $\mathcal{X}$, by detecting and avoiding it, without having to extend the syntax of the calculus. In contrast, notice that this is not possible for the λ-calculus.

A particular problem with α-conversion is that, although intuitively very clear, it is difficult to formalise, since it is normally expressed in global terms over a calculus. Normally, Barendregt's convention is quoted. Especially the notion of 'bound' varies greatly (e.g. x is considered bound not only in $\lambda x.M$, but also in M

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 2.1 (Reduction of $\mathbb{I}(\lambda x.xx)(\lambda y.y)_{\mathbb{I}}^{\lambda}$)

$$
\begin{array}{lll}
1. & \mathbb{I}(\lambda x.xx)(\lambda y.y)_{\mathbb{I}}^{\lambda} & \rightarrow \ (med),(exp\text{-}imp),(ren\not\curlywedge) \\
2. & (\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\widehat{x}(\langle x\cdot\delta\rangle\,\widehat{\delta}\,[x]\ \widehat{w}\langle w\cdot\alpha\rangle) & \rightarrow \\
 & & (\curlywedge a),(\curlywedge imp\text{-}out),(\curlywedge cap),(\curlywedge d),(exp) \\
3. & (\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\widehat{z}((\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\delta)\,\widehat{\delta}\,[z]\ \widehat{w}\langle w\cdot\alpha\rangle) & \rightarrow \ (exp\text{-}imp) \\
4. & ((\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\delta)\,\widehat{\delta}\dagger\widehat{y}\langle y\cdot\mu\rangle)\,\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \rightarrow \ (exp),(a\not\curlywedge) \\
5. & (\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\mu)\,\widehat{\mu}\not\curlywedge\widehat{w}\langle w\cdot\alpha\rangle & \rightarrow \ (exp\text{-}out\not\curlywedge) \\
6. & (\widehat{y}(\langle y\cdot\mu\rangle\,\widehat{\mu}\not\curlywedge\widehat{w}\langle w\cdot\alpha\rangle)\,\widehat{\mu}\cdot\eta)\,\widehat{\eta}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \rightarrow \ (d\not\curlywedge),(cap) \\
7. & (\widehat{y}\langle y\cdot\alpha\rangle\,\widehat{\mu}\cdot\eta)\,\widehat{\eta}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \rightarrow \ (exp) \\
8. & \widehat{y}\langle y\cdot\alpha\rangle\,\widehat{\mu}\cdot\alpha &
\end{array}
$$

This result is wrong; the error lies in step 5, where the rule $(exp\text{-}out\not\curlywedge)$ is erroneously applied. The side condition of that rule just checks if the plug occurs free *inside* the term, not if it is free in the left-hand side. The reduction should be:

$$
\begin{array}{lll}
5. & (\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\mu)\,\widehat{\mu}\not\curlywedge\widehat{w}\langle w\cdot\alpha\rangle & =_{\alpha} \\
6'. & (\widehat{y}\langle y\cdot\sigma\rangle\,\widehat{\sigma}\cdot\mu)\,\widehat{\mu}\not\curlywedge\widehat{w}\langle w\cdot\alpha\rangle & \rightarrow \ (d\not\curlywedge),(exp) \\
7'. & \widehat{y}\langle y\cdot\sigma\rangle\,\widehat{\sigma}\cdot\alpha & =_{\alpha} \ \widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\alpha
\end{array}
$$

This example is illustrated in Fig. 1.

2.1 Lazy copying of shared graphs [3]

The solution proposed in [3] for the above problem was to avoid, as for *λ-graphs*, the sharing of graphs that were involved in multiple cuts with other graphs. Similarly to the case for the λ-calculus [17], binding of connectors was 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 being to 'peel off a copy' of the graphs which might get affected by the double binding of connectors.

This method is similar to that of [17], but, as argued in many papers since then, this is too restrictive, in that it, even when done lazily, destroys a large amount of sharing. But perhaps the main objection to the solution of [3] is that it creates unnecessary overhead in that it invokes rebinding for *non-nested* double bindings of connectors, that created no conflict what-so-ever.

2.2 Preserving Barendregt's convention

After concluding [3], it was noticed that, as suggested above, the fact that a graph can be involved with two different cuts is not problematic. In fact, using term graph rewriting techniques, the particular λ-graph problem disappears; substitution 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, the necessary copying is done during the *building* 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, unlike for λ-calculus where the sharing in the graphs created the problem, here we are in reality dealing with the problem of α-conversion.

This is the problem addressed in this paper. The first solution we propose is the preserve Barendregt's convention on names, i.e. make sure that names never occur both free and bound. It is common practice to say that α-conversion is the machinery necessary to uphold Barendregt's convention, that states that an identifier should not appear both free and bound in a context (where a context can be a term, but also a type statement). It is especially the notion of *binding* that is important; for example, normally x is considered bound in all $\lambda x.M$, M

.

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

Definition 2.2 (α-SAFETY) We call a *circuit* ($\mathcal{X}$-graph) α-*safe* if it adheres to Barendregt's convention, i.e. no connector occurs both free and bound, and 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 circuit (graph) to an α-safe circuit (graph). We call a rewrite system α-safe if all its rules are α-safe.

For example, the circuit $(\widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}\cdot\mu)\,\widehat{\mu}\dagger\widehat{w}\langle w{\cdot}\alpha\rangle$ 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 $\mathcal{X}$, the rewrite rules that are *not* α-safe were identified. In Example 2.1, 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.

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 $\langle v{\cdot}\delta\rangle\,\widehat{\delta}\mathbin{\backslash\!\!\backslash}\widehat{y}P$ or $P\widehat{\beta}\mathbin{/\!\!/}\widehat{v}\langle v{\cdot}\delta\rangle$ to rename y by v, or β by δ respectively in P, where v,δ are fresh (see Lemma **??**). By activating the cuts, the renaming is forced to take place and stop other cuts from propagating over the renaming.

This effectively makes the extension of [3] via rp and rs obsolete; on the down-side, it is no longer possible to force eager or lazy evaluation of α-conversion without doing the same with the general propagation rules.

It has become clear that we should perform the α-conversion in rule (*exp-imp*). Let us consider the first choice (assume α and x are introduced):

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

In order to allow the rewrite to be executed like this, the side condition should express two extra criteria to avoid α-clashes: $y\neq z$, and $y\notin bs(P)$ (if either of the criteria do not hold, we have a nested binding to y on the right-hand side). If one of these last tests fails, renaming should take place. This implies that there are now two alternatives for this rule:

$$\begin{array}{lll}(\widehat{y}R\widehat{\beta}\cdot\alpha)\,\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) & \rightarrow & Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\dagger\widehat{z}P) \qquad\qquad y\neq z, y\notin bs(P)\\ (\widehat{y}R\widehat{\beta}\cdot\alpha)\,\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) & \rightarrow & Q\widehat{\gamma}\dagger\widehat{v}((\langle v{\cdot}\delta\rangle\,\widehat{\delta}\mathbin{\backslash\!\!\backslash}\widehat{y}R)\,\widehat{\beta}\dagger\widehat{z}P)\\ & & \qquad\qquad (y=z\vee y\in bs(P)), v, \delta\mathit{fresh}\end{array}$$

We could even add the alternative

$$(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) \;\to\; Q\widehat{\gamma}\dagger\widehat{v}(R\widehat{\beta}\dagger\widehat{z}P) \qquad y\notin fs(P), v\mathit{fresh}$$

Likewise, there are two alternatives for the second choice:

$$\begin{array}{rcll}(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) & \to & (Q\widehat{\gamma}\dagger\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P & \gamma\neq\beta\notin bp(Q)\\ (\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) & \to & (Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\not\leftarrow\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{z}P & \\ & & & (\beta=\gamma\vee\beta\in bp(Q)), v, \delta\mathit{fresh}\end{array}$$

Applying this solution to Example 2.1, we have, instead of the problematic step

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

the correction

$$\begin{array}{rcl}(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\gamma)\widehat{\gamma}\dagger\widehat{k}((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\;[k]\;\widehat{w}\langle w\cdot\alpha\rangle) & \to & (\mathit{exp\text{-}imp})\\ ((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}(\langle y\cdot\mu\rangle\widehat{\mu}\not\leftarrow\widehat{v}\langle v\cdot\beta\rangle))\widehat{\beta}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \to & (d\not\leftarrow)\\ ((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}(\langle y\cdot\mu\rangle\widehat{\mu}\dagger\widehat{v}\langle v\cdot\beta\rangle))\widehat{\beta}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \to & (\mathit{cap})\\ ((\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\delta)\widehat{\delta}\dagger\widehat{y}\langle y\cdot\beta\rangle)\widehat{\beta}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \to & (\mathit{exp})\\ (\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\beta)\widehat{\beta}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \to & (\mathit{exp})\\ \widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\alpha & &\end{array}$$

To guarantee α-safety, we need to do this for each rule where a possible α-conflict is introduced, like $(\nwarrow\mathit{cut})$:

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

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 circuit). With this in mind, the rule $(\nwarrow\mathit{cut})$ is amended with extra side conditions and extended with the following variants (where v,δ are *fresh*):

$$\begin{array}{rcll}P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \to & (P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\beta}\dagger\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R) & \beta\notin bp(P), y\notin bs(P)\\ P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \to & (P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\beta}\dagger\widehat{v}(P\widehat{\alpha}\nwarrow\widehat{x}(\langle v\cdot\delta\rangle\widehat{\delta}\nwarrow\widehat{y}R)) & \\ & & & \beta\notin bp(P), y\in bs(P)\\ P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \to & (P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\not\leftarrow\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R) & \\ & & & \beta\in bp(P), y\notin bs(P)\\ P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \to & (P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\not\leftarrow\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{v}(P\widehat{\alpha}\nwarrow\widehat{x}(\langle v\cdot\delta\rangle\widehat{\delta}\nwarrow\widehat{y}R)) & \\ & & & \beta\in bp(P), y\in bs(P)\end{array}$$

Almost all propagation rules (exceptions are $(d\not\leftarrow)$, $(\mathit{cap}\not\leftarrow)$, $(\nwarrow d)$, and $(\nwarrow\mathit{cap})$) should be treated like this; of the logical rules, only rule $(\mathit{exp\text{-}imp})$ needs dealing with as specified above. The advantage of this approach is that α-conversion itself is detected and dealt with:

Theorem 2.3 *Let $P\to_\alpha Q$ stand for the notion of rewriting on $\mathcal{X}$ obtained by changing the rules as above. Then: if P is α-safe, and $P\to_\alpha Q$, then Q is α-safe.*

The computational cost is low compared to the approach defined in [3] (see also Section 4); the price to pay is an increase in the number of rules.

2.3 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.

When letting go of Barendregt's convention, without the assumption of implicit α-conversion, the problem for variable-capture arises. Notice that this problem does not arise in terms that adhere to Barendregt's convention: free and bound connectors there are different, so capture is impossible. Instead, as discussed above, for those terms the problem is that of *nesting* of binders; to avoid this happening, renaming was used. Trying to avoid capture permits connectors to appear both free and bound; the needed modification of the rules is that they should detect possible capture of connectors.

Referring back to Example 2.1, the left-propagating cut in Step 5 should check if the connector μ *occurs free* in the left-hand term. This is not the case, so here α-conversion is necessary.

We will show we can always detect capturing safely, and perform α-conversion only then. The solution will, in appearance, be strikingly similar to that of Section 2.2 but for the fact that *freeness* is used rather than *boundness*, but, as argued in Section 4, it will show to be much more efficient; this is mainly because the solution of 2.2 α-converges circuits that are left untouched here.

The original idea for the solution presented in this section comes from the way α-conversion can be dealt with in the context of Bloo and Rose's calculus of explicit substitutions, $\lambda\mathbf{x}$ [10].

Definition 2.4 ($\lambda\mathbf{x}$ [10]) The set $\lambda\mathbf{x}$ is defined as follows:

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

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*.

Notice that the variable x is considered bound in $M\langle x := N\rangle$.

Definition 2.5 (Reduction in $\lambda\mathbf{x}$ [10]) The following reduction rules on $\lambda\mathbf{x}$ terms are identified.

$$\begin{array}{llll} (\lambda x.M)P & \to & M\langle x := P\rangle & (MN)\langle x := P\rangle \to M\langle x := P\rangle N\langle x := P\rangle \\ x\langle x := P\rangle & \to & P & (\lambda y.M)\langle x := P\rangle \to \lambda y.(M\langle x := P\rangle) \\ y\langle x := P\rangle & \to & y, \quad (y \neq x) & M\langle x := P\rangle \to M, \quad \text{if } x \notin fv(M) \end{array}$$

In fact, the two bottom rules overlap; this is similar to the situation in $\mathcal{X}$, with rules $(cap\not\prec)$ and $(gc\not\prec)$, and $(\not\setminus cap)$ and $(\not\setminus gc)$. 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, but now normally the problem is avoided altogether by considering terms 'modulo 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 an α-conflict while reducing can be expressed. In fact, variable capturing can be avoided radically by changing the reduction rule $(\lambda y.M)$

If one could use side-conditions on the rules, we could say:

$$(\lambda y.M)$$

Let us consider the first choice in rule $(\textit{exp-imp})$ (assume that α, x are *introduced*):

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

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 y in P; if $y \in fs(P)$, then the rule would bring that y under the binder $\hat{y}$ on the right-hand side, and renaming should take place. This implies that there are now four alternatives for the rule (*exp-imp*). First, for the first alternative (where again v, δ are fresh):

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

We could even add the alternative

$$(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) \;\to\; Q\widehat{\gamma}\dagger\widehat{v}(R\widehat{\beta}\dagger\widehat{z}P),\; y\notin fs(R), y\in fs(P)$$

Likewise, there are two alternatives for the second choice:

$$\begin{array}{lll}(\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) \;\to\; (Q\widehat{\gamma}\dagger\widehat{y}R)\widehat{\beta}\dagger\widehat{z}P, & \beta\notin fp(Q)\\ (\widehat{y}R\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P) \;\to\; (Q\widehat{\gamma}\dagger\widehat{y}(R\widehat{\beta}\nearrow\widehat{v}\langle v\cdot\delta\rangle))\widehat{\delta}\dagger\widehat{z}P, & \beta\in fp(Q)\end{array}$$

Also, since now we explicitly allow for connectors to occur both free and bound in a circuit, the rules need to check if the connector we try to connect to in a cut is actually really free. For example, rule $(\textit{exp-out}\nearrow)$ now becomes:

$$\begin{array}{lll}(\widehat{y}Q\widehat{\beta}\cdot\alpha)\widehat{\alpha}\nearrow\widehat{x}P \;\to\; (\widehat{y}(Q\widehat{\alpha}\nearrow\widehat{x}P)\widehat{\beta}\cdot\gamma)\widehat{\gamma}\dagger\widehat{x}P, & \alpha\in fp(Q)\;\&\;\alpha\neq\beta\\ (\widehat{y}Q\widehat{\beta}\cdot\alpha)\widehat{\alpha}\nearrow\widehat{x}P \;\to\; (\widehat{y}Q\widehat{\beta}\cdot\alpha)\widehat{\alpha}\dagger\widehat{x}P, & \alpha\notin fp(Q)\vee\alpha=\beta\end{array}$$

and rule $(\textit{exp})$ becomes:

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

The reduction of Example 2.1 should have been:

$$\begin{array}{lll}(\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\mu)\widehat{\mu}\nearrow\widehat{w}\langle w\cdot\alpha\rangle & \to(\textit{exp-out}\nearrow') & \\ (\widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\mu)\widehat{\mu}\dagger\widehat{w}\langle w\cdot\alpha\rangle & \to(\textit{exp}') & \widehat{y}\langle y\cdot\mu\rangle\widehat{\mu}\cdot\alpha\end{array}$$

Although the structure of these new rules is similar to those in Section 2.2, the improvement in execution speed is impressive, as can be seen in the last section.

3 Generalising $\mathcal{X}$

3.1 Optimising garbage collection

A natural optimisation is to check whether the renaming invoked by the alpha conversions is actually necessary, by checking if the connector to rename occurs free in the graph, thereby implicitly calling a garbage collection rule. The applicability of the garbage collection rules stated in Lemma **??** 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 rules; in fact, we will show their admissibility below (Theorem 3.3), for which we first need to show a number of results.

Lemma 3.1 *i) For all $\mathcal{X}$-circuits $R \equiv P\widehat{\alpha} \nleftarrow \widehat{x}Q$ with P, Q pure, there exists a reduction path $R \to R'$, with R' pure.*

ii) For all $\mathcal{X}$-circuits $R \equiv P\widehat{\alpha} \nrightarrow \widehat{x}Q$ with P, Q pure, there exists a reduction path $R \to R'$, with R' pure.

Proof: By induction on the depth d, of the active cut, where $d = 0$ if the cut does not appear in the circuit. □

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

Lemma 3.2 *For all $\mathcal{X}$-circuits P, there exists a reduction path $P \to P'$, with P' pure.*

We are now ready to justify the more general garbage collection rules:

Theorem 3.3 (Generalized garbage collection) *i)* $P\widehat{\alpha} \nleftarrow \widehat{x}Q \stackrel{\mathcal{X}}{=} P, \alpha \notin fp(P)$.

ii) $P\widehat{\alpha} \nrightarrow \widehat{x}Q \stackrel{\mathcal{X}}{=} Q, x \notin fs(Q)$.

Proof: *i*) The two cases to consider are: either P is *pure* or P is *not pure*.

(P *is* pure): This follows from rule $(gc\nleftarrow)$.

(P *is* not pure): Then by Lemma 3.2, there exists a reduction path from P to a circuit P', such that P' is pure. Note that this reduction path affects *only* the circuit P, thus, so we can reduce P to P' and also $P\widehat{\alpha} \nleftarrow \widehat{x}Q$ to $P'\widehat{\alpha} \nleftarrow \widehat{x}Q$. Since the reduction path does not introduce α in P', we can apply this first part of this proof, eliminating the outermost flagged cut.

ii) The justification for this rule is similar. □

This result now helps to justify more general deactivation rules.

Theorem 3.4 (Generalized deactivation) *i)* $P\widehat{\alpha} \nleftarrow \widehat{x}Q \stackrel{\mathcal{X}}{=} P\widehat{\alpha} \dagger \widehat{x}Q$ *, if P introduces α.*

ii) $P\widehat{\alpha} \nrightarrow \widehat{x}Q \stackrel{\mathcal{X}}{=} P\widehat{\alpha} \dagger \widehat{x}Q$ *, if Q introduces x.*

Proof: *i*) If P introduces α, we have two cases:

($P = \langle x \cdot \alpha \rangle$): By rule $(d\nleftarrow)$.

($P = \widehat{x}P'\widehat{\beta} \cdot \alpha$): Then $\alpha \notin fp(P)$, and

$$\begin{array}{rcl} (\widehat{x}P'\widehat{\beta} \cdot \alpha)\widehat{\alpha} \nleftarrow \widehat{x}Q & \to & (exp\text{-}out\nleftarrow) \\ (\widehat{x}(P'\widehat{\alpha} \nleftarrow \widehat{x}Q)\widehat{\beta} \cdot \gamma)\widehat{\gamma} \dagger \widehat{x}Q & \stackrel{\mathcal{X}}{=} & (3.3) \\ (\widehat{x}P'\widehat{\beta} \cdot \gamma)\widehat{\gamma} \dagger \widehat{x}Q & =_{\alpha} & (\widehat{x}P'\widehat{\beta} \cdot \alpha)\widehat{\alpha} \dagger \widehat{x}Q \end{array}$$

ii) If Q introduces x, we have two cases:

($Q = \langle x \cdot \beta \rangle$): By rule $(\nrightarrow d)$.

($Q = Q_1\widehat{\beta}\ [x]\ \widehat{y}Q_2$): Then $x \notin fs(Q_1, Q_2)$, and

$$\begin{array}{rcl} P'\widehat{\alpha} \nrightarrow \widehat{x}(Q_1\widehat{\beta}\ [x]\ \widehat{y}Q_2) & \to & (\nrightarrow imp\text{-}out) \\ P'\widehat{\alpha} \dagger \widehat{v}((P'\widehat{\alpha} \nrightarrow \widehat{x}Q_1)\widehat{\beta}\ [v]\ \widehat{y}(P'\widehat{\alpha} \nrightarrow \widehat{x}Q_2)) & \stackrel{\mathcal{X}}{=} & (3.3) \\ P'\widehat{\alpha} \dagger \widehat{v}(Q_1\widehat{\beta}\ [v]\ \widehat{y}Q_2) & =_{\alpha} & P'\widehat{\alpha} \dagger \widehat{x}(Q_1\widehat{\beta}\ [x]\ \widehat{y}Q_2) \end{array}$$

Adding these more generic reduction rules gave another significant improvement, on which we report in Section 4.

3.2 Available connectors

Another optimisation of the reduction system that we will present in this section is based on the notion of *availability*, as first defined in [12], where strong normalisation of reduction is studied for $\lambda\mathbf{x}$ using intersection types. An important role is played by the notion of *available* variable in a term, which is a generalization of the classical notion of free variable.

Definition 3.5 The set of *available sockets* of a circuit is defined by:

$$\begin{array}{lcllcl} as(\langle x{\cdot}\alpha\rangle) & = & \{x\} & as(P\widehat{\alpha}\,[y]\,\widehat{x}Q) & = & as(P)\cup\{y\}\cup as(Q)\setminus\{x\} \\ as(\widehat{y}P\widehat{\beta}{\cdot}\alpha) & = & as(P)\setminus\{y\} & as(P\widehat{\alpha}\dagger\widehat{x}Q) & = & as(P)\cup as(Q)\setminus\{x\} \end{array}$$

$$as(P\widehat{\alpha}\nwarrow\widehat{x}Q) = \begin{cases} as(P)\cup as(Q)\setminus\{x\} & x\in as(Q) \\ as(Q) & x\notin as(Q) \end{cases}$$

$$as(P\widehat{\alpha}\nearrow\widehat{x}Q) = \begin{cases} as(P)\cup as(Q)\setminus\{x\} & \alpha\in ap(P) \\ as(P) & \alpha\notin ap(P) \end{cases}$$

The set of *available plugs*, $ap(P)$ of a circuit P is defined similarlyby:

$$\begin{array}{lcllcl} ap(\langle x{\cdot}\alpha\rangle) & = & \{\alpha\} & ap(P\widehat{\alpha}\,[y]\,\widehat{x}Q) & = & ap(P)\cup ap(Q)\setminus\{\alpha\} \\ ap(\widehat{y}P\widehat{\beta}{\cdot}\alpha) & = & ap(P)\cup\{\alpha\}\setminus\{\beta\} & ap(P\widehat{\alpha}\dagger\widehat{x}Q) & = & ap(P)\cup ap(Q)\setminus\{\alpha\} \end{array}$$

$$ap(P\widehat{\alpha}\nwarrow\widehat{x}Q) = \begin{cases} ap(P)\cup ap(Q)\setminus\{\alpha\} & x\in as(Q) \\ ap(Q) & x\notin as(Q) \end{cases}$$

$$ap(P\widehat{\alpha}\nearrow\widehat{x}Q) = \begin{cases} ap(P)\cup ap(Q)\setminus\{\alpha\} & \alpha\in ap(P) \\ ap(P) & \alpha\notin ap(P) \end{cases}$$

First of all, much in the style of the results of Subsection 3.1, we can show that it is sound to extend the reduction relation with the cases for garbage collection expressed using available connectors, rather than free.

We first show the following lemma, which shows the commutativity of activated cuts:

Lemma 3.6
$$\begin{array}{rcl} (P\widehat{\gamma}\nearrow\widehat{y}Q)\widehat{\alpha}\nearrow\widehat{x}R & \stackrel{\mathcal{X}}{=} & (P\widehat{\alpha}\nearrow\widehat{x}R)\widehat{\gamma}\nearrow\widehat{y}(Q\widehat{\alpha}\nearrow\widehat{x}R) \\ (P\widehat{\gamma}\nwarrow\widehat{y}Q)\widehat{\alpha}\nearrow\widehat{x}R & \stackrel{\mathcal{X}}{=} & (P\widehat{\alpha}\nearrow\widehat{x}R)\widehat{\gamma}\nwarrow\widehat{y}(Q\widehat{\alpha}\nearrow\widehat{x}R) \\ P\widehat{\gamma}\nwarrow\widehat{y}(Q\widehat{\alpha}\nwarrow\widehat{x}R) & \stackrel{\mathcal{X}}{=} & (P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\gamma}\nwarrow\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R) \\ P\widehat{\gamma}\nwarrow\widehat{y}(Q\widehat{\alpha}\nearrow\widehat{x}R) & \stackrel{\mathcal{X}}{=} & (P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\gamma}\nearrow\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R) \end{array}$$

Proof: Straightforward, by induction on the structure of circuits.

Theorem 3.7 *i)* $P\widehat{\alpha}\nearrow\widehat{x}Q \stackrel{\mathcal{X}}{=} P, \alpha\notin ap(P)$.

ii) $P\widehat{\alpha}\nwarrow\widehat{x}Q \stackrel{\mathcal{X}}{=} Q, x\notin as(Q)$.

Proof: By induction on the structure of terms. The cases for $P=\langle x{\cdot}\beta\rangle, \widehat{y}P'\widehat{\gamma}{\cdot}\beta$, or $Q_1\widehat{\gamma}\,[x]\,\widehat{y}Q_2$ are straightforward (by induction). As for the cuts, the case for the unactivated cut follows by the corresponding propagation rule (either $(\nwarrow cut)$ or $(cut\nearrow)$) and induction, and the case for activated cuts follows from Lemma 3.6 and induction.

By the above result, we can weaken our side-conditions on the rules and check for availability rather than free-ness. We know that the connectors that are not available but free will disappear during reduction, as will the circuits that will be connected via them; such connections are redundant and would only slow down the reduction engine, not affecting reachable normal forms. Using availability allows more sub-circuits to be considered garbage, thereby optimising reduction.

We modify the logical rules in Definition **??**, and generalised rules of Section 3.1, replacing free connector checks with availability checks. Additionally, we attach a side condition to each rule in Definition **??** to permit propagation only if the socket (plug) involved in the right-cut (left-cut) is available in the corresponding sub-circuit.

With that in mind, the definitions below are a variant of the rules defined in [2], that define an optimisation of the reduction relation, and depend on a notion of *connectability*:

Definition 3.8 Connectability of a connector in a circuit is defined by:

(x *is connectable in* P): Either $P = P\widehat{\alpha}[x]\widehat{y}Q$, and x is not available in P, Q, or $P = \langle x{\cdot}\delta\rangle$.
(δ *is connectable in* P): Either $P = \widehat{x}Q\widehat{\beta}{\cdot}\delta$, and δ is not available in Q, or $P = \langle x{\cdot}\delta\rangle$.

Notice that predicate essentially states that the connector has only one significant occurrence on the outermost level.

Definition 3.9 (Reduction: **Logical rules**) Reduction of well-connected cuts, i.e., where both connectors mentioned in the cut are connectable, is defined by:

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

Definition 3.10 (Enabling) We define the following two enabling rules.

$$\begin{array}{rrcll}
(a\not\prec): & P\widehat{\alpha} \dagger \widehat{x}Q & \to & P\widehat{\alpha} \not\prec \widehat{x}Q & \textit{if } \alpha \textit{ not connectable in } P \\
(\not\succ a): & P\widehat{\alpha} \dagger \widehat{x}Q & \to & P\widehat{\alpha} \not\succ \widehat{x}Q & \textit{if } x \textit{ not connectable in } Q
\end{array}$$

The enabled cuts are different from the active cuts in that no longer the dagger is tilted only if the connector is unique and only occurring at the outermost level. We now also allow tilting if the connector is 'out of reach', i.e. present, but in a circuit that will certainly be discarded.

Notice that a cut $(\widehat{y}P\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\gamma}[x]\widehat{z}P)$ can be enabled if either the plug α is available in P, or the socket x is available in Q or R; otherwise, the logical rule (cut) applies. This change in activation, and the fact that availability rather than freenees is used in the logical rules, causes a different formulation of the propagation rules.

Definition 3.11 (Left propagation)

$$\begin{array}{rcll}
Q\widehat{\alpha} \not\prec \widehat{x}P & \to & Q\widehat{\alpha} \dagger \widehat{x}P, & \alpha \textit{connectable in } Q \\
Q\widehat{\alpha} \not\prec \widehat{x}P & \to & Q, & \alpha \notin ap(Q) \\
(\widehat{y}Q\widehat{\beta}{\cdot}\alpha)\widehat{\alpha} \not\prec \widehat{x}P & \to & (\widehat{y}(Q\widehat{\alpha} \not\prec \widehat{x}P)\widehat{\beta}{\cdot}\gamma)\widehat{\gamma} \dagger \widehat{x}P, & \gamma \textit{ fresh}, \alpha \in ap(Q) \\
(\widehat{y}Q\widehat{\beta}{\cdot}\gamma)\widehat{\alpha} \not\prec \widehat{x}P & \to & \widehat{y}(Q\widehat{\alpha} \not\prec \widehat{x}P)\widehat{\beta}{\cdot}\gamma, & \gamma \neq \alpha \in ap(Q) \\
(Q\widehat{\beta}[z]\widehat{y}R)\widehat{\alpha} \not\prec \widehat{x}P & \to & (Q\widehat{\alpha} \not\prec \widehat{x}P)\widehat{\beta}\ [z]\ \widehat{y}(R\widehat{\alpha} \not\prec \widehat{x}P), & \alpha \in ap(Q,R) \\
(Q\widehat{\beta} \dagger \widehat{y}R)\widehat{\alpha} \not\prec \widehat{x}P & \to & (Q\widehat{\alpha} \not\prec \widehat{x}P)\widehat{\beta} \dagger \widehat{y}(R\widehat{\alpha} \not\prec \widehat{x}P), & \alpha \in ap(Q,R)
\end{array}$$

Definition 3.12 (Right propagation)

$$\begin{array}{lcll}
P\widehat{\alpha} \not\succ \widehat{x}Q & \to & P\widehat{\alpha} \dagger \widehat{x}Q, & x \textit{connectable in } Q \\
P\widehat{\alpha} \not\succ \widehat{x}Q & \to & Q, & x \notin as(Q) \\
P\widehat{\alpha} \not\succ \widehat{x}(\widehat{y}Q\widehat{\beta}{\cdot}\gamma) & \to & \widehat{y}(P\widehat{\alpha} \not\succ \widehat{x}Q)\widehat{\beta}{\cdot}\gamma, & x \in as(Q) \\
P\widehat{\alpha} \not\succ \widehat{x}(Q\widehat{\beta}[x]\widehat{y}R) & \to & P\widehat{\alpha} \dagger \widehat{z}((P\widehat{\alpha} \not\succ \widehat{x}Q)\widehat{\beta}\ [z]\ \widehat{y}(P\widehat{\alpha} \not\succ \widehat{x}R)), & \\
 & & & z \textit{ fresh}, x \in as(Q,R) \\
P\widehat{\alpha} \not\succ \widehat{x}(Q\widehat{\beta}[z]\widehat{y}R) & \to & (P\widehat{\alpha} \not\succ \widehat{x}Q)\widehat{\beta}\ [z]\ \widehat{y}(P\widehat{\alpha} \not\succ \widehat{x}R), & z \neq x \in as(Q,R) \\
P\widehat{\alpha} \not\succ \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R) & \to & (P\widehat{\alpha} \not\succ \widehat{x}Q)\widehat{\beta} \dagger \widehat{y}(P\widehat{\alpha} \not\succ \widehat{x}R), & x \in as(Q,R)
\end{array}$$

We write $\to_o$ for the (transitive, compatible) reduction relation generated from these rules.

Of course, before implementing this system, also the capture-avoidance of the previous section needs to be applied and checked; notice α-conversion would only be performed on *available* connectors, though. This is left for future work.

Term	Lazy-copying		α-safety		α-safety (Gen.)		Capture (Gen.)	
22II	11901	8939	3730	3370	2282	2178	2427	1233
222II	...	...	80695	35051	15636	7785	8949	3548
2222II	...	...	257179	78242	45294	16137	40511	6132
22222II	...	...	263528	79717	45657	16466	40924	6461
210II	3987	3226	5120	2163	2029	1685	1776	1148
2210II	107627	19182	13080	9280	5420	4977	4735	2690
22210II	...	...	180153	43292	25932	9303	13909	2875
Q(n=2)	...	...	23207	79401	8617	9053	6870	4212
Q(n=5)	...	...	30743	122391	12664	10955	7701	5043
Q(n=10)	...	...	44663	201521	15056	14125	9086	6428
Q(n=20)	...	...	77603	387831	23125	20465	11856	9198
Q(n=50)	...	...	217223	...	47305	39485	20166	17508

Figure 2: CBV (*left*) and CBN (*right*) benchmarks

4 Measurements

In this section we compare the efficiency of the solutions presented in this paper using the term-graph rewriting tool. The tool is written Java and is based on standard term-graph rewriting techniques, with support for conditional and higher-order rewriting.

We choose to measure the running of (interpreted) lambda terms since these are well-known benchmarks [1, 14], and the efficiency of the various formalisms and abstract machines can be better compared. We can of course not confront (published) run-time measurements because of differences in platforms and processor speeds. It could be interesting to measure the running of circuits that are not interpreted lambda terms, but we would have no means to compare those to other implementations.

We use the usual encoding for Church Numerals ($\mathsf{n} = \lambda xy.x^n y$). In addition, we use the combinators, $\mathsf{Q} = (\lambda z.(\lambda x.zxxx)(\lambda y.\mathsf{2}(\lambda x.y(x\mathsf{I}))n))\mathsf{II}$, with n replaced with a chosen Church Numeral and $\mathsf{I} = \lambda x.x$. The interpretation function from λ-calculus to $\mathcal{X}$ [2] (Definition **??**) is used to encode each benchmark before input in to the tool. We count the number of *add* and *remove* operations for nodes and edges, together with the number of edge redirections. For each system, we test two strategies: call-by-value and call-by-name.

Figure 2 lists our results, where terms taking more than 500,000 interactions to normalise are omitted. Our experimental results emphasise the unnecessary restrictions imposed while adhering to Barendregt's convention, and show the importance of the generalised rules for any efficient implementation of $\mathcal{X}$.

Our various attempts to find efficient α-conversion solutions also involved optimisations of the tool. The α-conversion variants for each rule and the use of side-conditions requiring subgraph traversals created a bottleneck during the matching phase of the rewrite procedure. An initial parse of the rewrite rules groups together rules whose *LHS* are structurally equivalent. We also use labelled graphs to store and reuse information about the freeness of connectors. The most optimal solution to date combines the notion of availability together with the avoiding capture solution. Considering a term like, $(\widehat{x}(\langle x{\cdot}\delta\rangle\widehat{\alpha}\not\curlywedge\widehat{y}\langle y{\cdot}\beta\rangle)\,\widehat{\mu}{\cdot}\gamma)\,\widehat{\beta}\not\curlywedge\widehat{z}P$, we observe that β is free, but *not* available; thus propagation of the outermost cut is permitted using the freeness definition of garbage collection. Using the availability solution, ($gc\not\curlywedge$) is applicable, side-stepping an unnecessary reduction.

Conclusions and future work

We have studied various solutions for the problem of α-conversion 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. We aim to show that the reduction strategy used in the tool (essentially left-most outer-most) is normalising. We also aim to show that the availability check actually improves reduction; our measurements so far do not indicate much improvement, but this seems to be due to the reduction strategy used.

References

[1] A. Asperti, C. Giovanetti, and A. Naletto. The bologna optimal higher-order machine. *J. Funct. Program.*, 6(6):763–810, 1996.

[2] S. van Bakel, S. Lengrand, and P. Lescanne. The language $\mathcal{X}$: circuits, computations and classical logic. In *ICTCS'05*, LNCS 3701, pages 81–96, 2005.

[3] S. van Bakel and J. Raghunandan. Implementing $\mathcal{X}$. In *TermGraph'04*, ENTCS, 2005.

[4] S. van Bakel, J. Raghunandan, and A. Summers. Term Graphs, α-conversion and Principal Types for $\mathcal{X}$. *Manuscript*, 2005.

[5] F. Barbanera and S. Berardi. A symmetric lambda calculus for classical program extraction. *Information and Computation*, 125(2):103–117, 1996.

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

[7] 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 *PARLE*, LNCS 259-II:141–158, 1987.

[8] 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 *PARLE*, LNCS 259-II:159–175, 1987.

[9] E. Barendsen and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. *Mathematical Structures of 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*, pages 62–72, 1995.

[11] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP'00*, pages 233–243. ACM, 2000.

[12] S. Lengrand, P. Lescanne, D. Dougherty, M. Dezani-Ciancaglini, and S. van Bakel. Intersection types for explicit substitutions. *Information and Computation*, 189(1):17–42, 2004.

[13] Stéphane Lengrand. Call-by-value, call-by-name, and strong normalization for the classical sequent calculus. In ENTCS 86, 2003.

[14] Ian Mackie. Efficient lambda-evaluation with interaction nets. In *RTA*, pages 155–169, 2004.

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

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

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