Term Graphs, α-conversion and Principal Types for $\mathcal{X}$

Steffen van Bakel[†] Jayshan Raghunandan Alexander Summers

Department of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ, UK,
{svb,jr200,ajs300m}@doc.ic.ac.uk

Abstract

In this paper we study the calculus of circuits $\mathcal{X}$, as first presented in [13] and studied in detail in [4]. We will present a number of new implementations of $\mathcal{X}$ using term graph rewriting techniques, which are improvements of the technique used in [5]. We show that alpha conversion can be dealt with 'on the fly', and that explicit copying can be avoided, by o presenting, discussing and comparing solutions to these problems.
We will define a notion of type assignment on circuits by labelling input and output connectors with types and show that this system has a principal contexts property, which generalises principal types. We extend the system, defining a non-standard notion of type assignment on term-graph rewriting. Finally, we show that types are preserved by interpretation to graphs and reduction.

Introduction

This paper will present a notion of decidable type assignment for the (untyped) calculus $\mathcal{X}$, as well as an improved term graph rewriting model of implementation. $\mathcal{X}$ is a new calculus, that was defined in [13] and was later extensively studied in [4]; in [2] a notion of type assignment with intersection and union types was presented. $\mathcal{X}$ is defined as a substitution-free programming language that, perhaps surprisingly, is extremely well-equipped to describe the behaviour of functional programming languages at a very low level of granularity (see [4]).

The origin of $\mathcal{X}$ lies within the quest for a language designed to give a Curry-Howard correspondence with the sequent calculus for Classical Logic, in particular that of [17]. In the past, this topic has been mainly studied in the context of *natural deduction systems*. Natural Deduction offers an intuitive method of reasoning about Classical Logic, which is designed to correspond closely with the arguments commonly used in mathematical proofs. However, from a logical point of view, the Sequent Calculus is generally accepted to have nicer symmetries and be more amenable to the proof of technical theorems. The originator of both Natural Deduction and Sequent Calculus, Gerhard Gentzen, commented that even his intuitionistic Natural Deduction calculus "lacks a certain formal elegance", while its Classical counterpart fares still worse, breaking the symmetry between the introduction and elimination rules [16]. Because of these technical difficulties, Gentzen found that to achieve the main results of [12], "I had to provide a logical calculus especially suited to the purpose. For this the natural [deduction] calculus proved unsuitable." The calculus described is the Sequent Calculus, and the main result is commonly referred to as the *Haupsatz*, which states that any sequent calculus proof using the special *cut rule* may be reduced to a "cut-free" proof with the same conclusion sequent. It is this process of *cut-elimination* (and the corresponding notion of *normalisation* of Natural Deduction proofs) which has sparked much of the research

[†] Partially supported by the MIKADO project of the IST-FET Global Computing Initiative, no IST-2001-32222

into a correspondence between computation and Classical Logic.

Starting from different approaches in that area [11, 17], in [13] the calculus $\mathcal{X}$ was introduced, and shown to be equivalent to the $\overline{\lambda}\mu\tilde{\mu}$-calculus of [11] in terms of expressiveness. Using this correspondence, a strong normalisation result was shown for $\overline{\lambda}\mu\tilde{\mu}$. We should point out that [13], as well as [17], did not study any property of *untyped* $\mathcal{X}$, but focused only on its typed aspects in connection with the sequent calculus. [17] set out to study the structure of proofs, so the terms there carry types and correspond to proofs. In $\mathcal{X}$, we study terms *without* types, and drop the condition that terms should represent proofs of the logic altogether: we study a pure calculus.

While studying $\mathcal{X}$ as an untyped language, it became apparent that $\mathcal{X}$ provides an excellent general purpose machine, very well suited to encode various calculi (for details, see [4]). Amongst the calculi studied in that paper, the Calculus of Explicit Substitutions, $\lambda\mathbf{x}$, stands out. In fact, a 'subatomic' level was reached by decomposing explicit substitutions into smaller components. Even more, the calculus is actually symmetric [7]; 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 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. 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 a notion 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 calculus and, more importantly, with the reduction engine. A tool[1] was developed using the term graph rewriting technology, that allows users to not only input circuits from $\mathcal{X}$, but also terms from the λ-calculus, using the interpretation of the latter into $\mathcal{X}$ as specified in [4]. In [5] 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 discuss how to improve on the choices made in [5].

One of the goals of our work is to come to the definition of a programming language based on $\mathcal{X}$; to that end, we sought to implement the notion of type assignment as first defined in [4]. This notion is non-standard (at least in the context of functional programming) in that the syntactic objects of the language are not all 'assigned' a type; as in the case of the π-calculus only the connectors, which act as input and output channels, carry types. To achieve this we defined a notion of *principal context* which generalises principal types. In this paper, we show that type assignment is decidable, even in this non-standard language with its non-standard notion of type assignment.

1 The calculus $\mathcal{X}$

In this section we will give the definition of the $\mathcal{X}$-calculus that was proven to be a fine-grained implementation model for various well-known calculi in [4]. It features two separate

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

categories of 'connectors', *plugs* and *sockets*, that act as input and output channels, and is defined without any notion of substitution.

Definition 1.1 (SYNTAX) The circuits of the $\mathcal{X}$-calculus are defined by the following syntax, where x, y range over the infinite set of *sockets*, α, β over the infinite set of *plugs*.

$$P, Q ::= \langle x{\cdot}\alpha \rangle \mid \widehat{y} P \widehat{\beta}{\cdot}\alpha \mid P \widehat{\beta}\,[y]\,\widehat{x} Q \mid P \widehat{\alpha} \dagger \widehat{x} Q$$

We use the following terminology for circuits: $\langle x{\cdot}\alpha \rangle$ is called a *capsule*, $(\widehat{y} P \widehat{\beta}{\cdot}\alpha)$ an *export* circuit, $(P \widehat{\beta}\,[y]\,\widehat{x} Q)$ a *mediator* and $(P \widehat{\alpha} \dagger \widehat{x} Q)$ a *cut*.

The $\widehat{\cdot}$ symbolises that the socket or plug underneath is bound in the circuit. For example, occurrences of β in P and of x in Q are bound in $(P \widehat{\beta}\,[y]\,\widehat{x} Q)$. The notion of bound and free connector is defined as usual, , and we will identify circuits that only differ in the names of bound connectors (modulo α-conversion, as usual). Of course α-conversion needed to be dealt with explicitly at the level of the tool; this will be discussed extensively in Section 3.

The calculus, defined by the reduction rules below, explains in detail how cuts are propagated through circuits to be eventually evaluated at the level of capsules. Reduction is defined by specifying both the interaction between well-connected basic syntactic structures, and how to deal with propagating active nodes to points in the circuit where they can interact.

It is important to know when a connector is introduced, i.e. is connectable, i.e. is exposed and unique; this will play an important role in the reduction rules. Informally, a circuit P introduces a socket x if P is constructed from sub-circuits which do not contain x as free socket, so x only occurs at the "top level." This means that P is either a mediator with a middle connector $[x]$ or a capsule with left part x. Similarly, a circuit introduces a plug α if it is an export that "creates" α or a capsule with right part α.

Definition 1.2 (INTRODUCTION) (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 $P = \langle x{\cdot}\alpha \rangle$.

The main reduction rules are defined by:

Definition 1.3 (LOGICAL RULES) Assume both α and x to be introduced in, respectively, the left- and right-hand side of the main cuts below.

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

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 mediator via the socket x. The effect of the reduction will be that the exported function is placed in-between the two sub-terms of the mediator, acting as interface between them.

We now need to define how to reduce a cut when one of its sub-circuits does not introduce a connector mentioned in the cut.

Definition 1.4 (ACTIVE CUTS) The syntax is extended with two *flagged* or *active* cuts:

$$P ::= \ldots \mid P_1 \widehat{\alpha} \nearrow \widehat{x} P_2 \mid P_1 \widehat{\alpha} \nwarrow \widehat{x} P_2$$

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

We define two cut-activation rules.

$$\begin{array}{rl} (a\not\leftarrow): & P\widehat{\alpha} \dagger \widehat{x}Q \;\to\; P\widehat{\alpha} \not\leftarrow \widehat{x}Q \;\; \textit{if } P \textit{ does not introduce } \alpha \\ (\nwarrow a): & P\widehat{\alpha} \dagger \widehat{x}Q \;\to\; P\widehat{\alpha} \nwarrow \widehat{x}Q \;\; \textit{if } Q \textit{ does not introduce } x \end{array}$$

The activated cuts (obtained from cuts to which the logical rules cannot be applied) are introduced to obtain a fine-tuned reduction system. An activated cut is processed by 'pushing' it systematically through the syntactic structure of the circuit in the direction indicated by the tilting of the dagger, to reach a position where a logical rule can be applied.

Definition 1.5 (Propagation rules) Left propagation:

$$\begin{array}{rrcll} (d\not\leftarrow): & \langle y\cdot\alpha\rangle\widehat{\alpha} \not\leftarrow \widehat{x}P & \to & \langle y\cdot\alpha\rangle\widehat{\alpha} \dagger \widehat{x}P & \\ (cap\not\leftarrow): & \langle y\cdot\beta\rangle\widehat{\alpha} \not\leftarrow \widehat{x}P & \to & \langle y\cdot\beta\rangle, & \beta \neq \alpha \\ (exp\text{-}out\not\leftarrow): & (\widehat{y}Q\widehat{\beta}\cdot\alpha)\widehat{\alpha} \not\leftarrow \widehat{x}P & \to & (\widehat{y}(Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\cdot\gamma)\widehat{\gamma} \dagger \widehat{x}P, & \gamma \textit{ fresh} \\ (exp\text{-}in\not\leftarrow): & (\widehat{y}Q\widehat{\beta}\cdot\gamma)\widehat{\alpha} \not\leftarrow \widehat{x}P & \to & \widehat{y}(Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\cdot\gamma, & \gamma \neq \alpha \\ (imp\not\leftarrow): & (Q\widehat{\beta}[z]\widehat{y}R)\widehat{\alpha} \not\leftarrow \widehat{x}P & \to & (Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta}\,[z]\;\widehat{y}(R\widehat{\alpha} \not\leftarrow \widehat{x}P) & \\ (cut\not\leftarrow): & (Q\widehat{\beta} \dagger \widehat{y}R)\widehat{\alpha} \not\leftarrow \widehat{x}P & \to & (Q\widehat{\alpha} \not\leftarrow \widehat{x}P)\widehat{\beta} \dagger \widehat{y}(R\widehat{\alpha} \not\leftarrow \widehat{x}P) & \end{array}$$

Right propagation:

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

We write $\to$ for the (reflexive, transitive, compatible) reduction relation generated by the logical, propagation and activation rules.

The reduction relation $\to$ is not confluent; this comes in fact from the critical pair that activates a cut $P\widehat{\alpha} \dagger \widehat{x}Q$ in two ways if P does not introduce α and Q does not introduce x. In [4], two sub-reduction systems were introduced which explicitly favour one kind of activation whenever the above critical pair occurs:

Definition 1.6 (Call-by-name and Call-by-value) We define Call-by-name and Call-by-value reduction by:

- if a cut can be activated in two ways, CBV can only activate it via $(a\not\leftarrow)$; we write $P \to_{\mathrm{V}} Q$ in that case.
- CBN only activates such a cut via $(\nwarrow a)$; similarly, we write $P \to_{\mathrm{N}} Q$.

In [4] some basic properties were shown, which essentially show that the calculus is well-behaved, as well as the relation between $\mathcal{X}$ and a number of other calculi. These results motivate the formulation of admissible rules:

Lemma 1.7 (Garbage Collection and Renaming)

$$\begin{array}{rrcll} (gc\not\leftarrow): & P\widehat{\alpha} \not\leftarrow \widehat{x}Q & \to & P & \textit{if } \alpha \notin fp(P), P \textit{ pure} \\ (\nwarrow gc): & P\widehat{\alpha} \nwarrow \widehat{x}Q & \to & Q & \textit{if } x \notin fs(Q), Q \textit{ pure} \\ (ren\not\leftarrow): & P\widehat{\delta} \dagger \widehat{z}\langle z\cdot\alpha\rangle & \to & P[\alpha/\delta] & P \textit{ pure} \\ (\nwarrow ren): & \langle z\cdot\alpha\rangle\widehat{\alpha} \dagger \widehat{x}P & \to & P[z/x] & P \textit{ pure} \end{array}$$

$$
\begin{array}{rcl}
\llbracket x \rrbracket & = & (x \mid \emptyset) \\
\llbracket \alpha \rrbracket & = & (\alpha \mid \emptyset) \\
\llbracket \langle x \cdot \alpha \rangle \rrbracket & = & (r \mid \{r : \mathsf{cap}(r_1, r_2)\} \cup G_1 \cup G_2), \\
\text{where} & & (r_1 \mid G_1) = \llbracket x \rrbracket \\
& & (r_2 \mid G_2) = \llbracket \alpha \rrbracket
\end{array}
$$

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

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

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

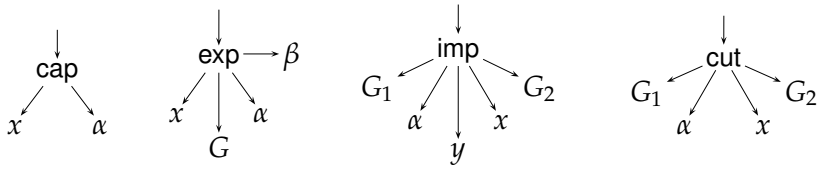
Figure 1: Interpreting circuits as graphs

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

In [5], a first implementation of $\mathcal{X}$ using term graph rewriting techniques was presented. The choice to use term graph rewriting, both in that paper and here, comes mainly from the observation that the reduction rules for $\mathcal{X}$ form a *conditional term rewriting system*. This might seem surprising, seen that the origin of $\mathcal{X}$ is a logic, and one could expect it to be close to λ-calculus. The non-standard aspects this rewrite system are a notion of binding, and that the rewrite rules are defined using *three* different classes of open (variable) nodes (for plugs, sockets, and circuits). Since for a general term rewriting system the term-graph technology is the best platform for implementation, this observation prompted us to attempt an implementation of $\mathcal{X}$ using this framework.

The technique applied is the standard one of [15, 8, 9], where terms and rewrite rules are *lifted* to graphs; it uses the *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. We now define a notion of term graphs:

Definition 2.1 Using the signature $\{\mathsf{cap}, \mathsf{cut}, \mathsf{cutl}, \mathsf{cutr}, \mathsf{exp}, \mathsf{med}\}$, we define (ordered) graphs by:



The last alternative has also variants with cutl and cutr in place of cut.

It should be noted that, 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.

This leads to the following formal definition of interpreting circuits in $\mathcal{X}$ as graphs (using the notation of [10]).

Definition 2.2 (Graph interpretation) For each circuit P, its *graph interpretation*, $\llbracket P \rrbracket$, expressed as a pair consisting of a label for the *root* of the graph and a set of *edges* is defined in Figure 1.

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. Sharing bound connectors would not be a problem for the reduction engine, but would create problems for the properties we want to show for the notion of type assignment of Section 4.

Definition 2.3 ($\mathcal{X}$-GRAPHS AND TERM GRAPH REWRITE RULES) *i*) We define the set of *initial* $\mathcal{X}$-graphs as the image of the circuits of $\mathcal{X}$ under $\llbracket \cdot \rrbracket$.

ii) The lifting of the $\mathcal{X}$ reduction rules to term graph rewriting rules is expressed by:

$$\llbracket \mathit{left} \to \mathit{right} \rrbracket \;=\; (r_l \mid G_l \cup G_r) \quad \text{where } (r_l \mid G_l) \;=\; \llbracket \mathit{left} \rrbracket \qquad (r_r \mid G_r) \;=\; \llbracket \mathit{right} \rrbracket$$

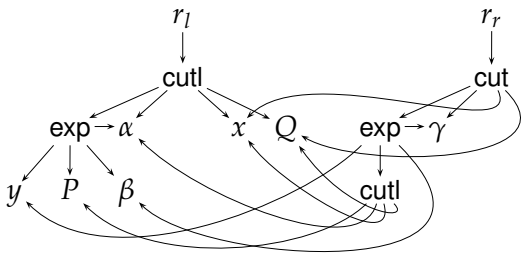
These rules induce a notion $G \to_{\mathrm{G}} G'$ of term graph rewriting.

iii) We define the set of $\mathcal{X}$-graphs as closure under $\to_{\mathrm{G}}$ of initial $\mathcal{X}$-graphs.

Notice that since $\llbracket x \rrbracket = (x \mid \emptyset)$, and we can assume this to be unique, the left and right-hand side graphs for a rule are joined on the connectors.

Example 2.1 The result of lifting rule $(\mathit{exp\text{-}out}\not\star)$ to a term graph rule becomes:

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



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

Definition 2.4 *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.

This definition depends clearly on the *in-degree*, the number of arcs going into a node.

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

We now have the following adequacy result:

Theorem 2.2 *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, if and only if so is P_2.*

A special property of the interpretation $\llbracket \cdot \rrbracket$ is that *all* occurrences of a connector x, say, are mapped onto a single node, whether or not this was bound in two or more different subterms. This might perhaps create suspicion, and is indeed something we will choose to avoid when dealing with types, but, by the very fact that $\mathcal{X}$ is *not* a substitution calculus, the reduction engine is indifferent to this. The main reason is that this non-nested sharing of binders does not violate Barendregt's convention of free and bound identifiers.

We can now prove the following result:

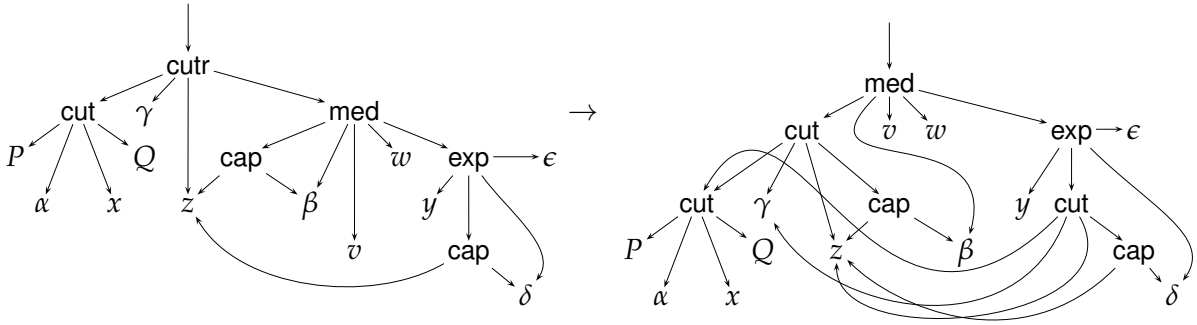
Theorem 2.3 *If $P \to Q$ in one step, then there exists an $\mathcal{X}$-graph G such that: $\llbracket P \rrbracket \to_{\mathrm{G}} G$, and Unrv $G = \llbracket Q \rrbracket$.*

Notice that, by the non-confluent character for $\mathcal{X}$, and the sharing used in the term-graph rewriting engine, we cannot prove a similar result for many-step reduction paths, as illustrated by the following.

Example 2.4 Let P and Q be such that $\alpha \notin fp(P)$ and $x \notin fp(Q)$, so $P \leftarrow P\widehat{\alpha} \dagger \widehat{x}Q \rightarrow Q$. Now (assume $z \neq v$):

$$\begin{array}{lll}
(P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\gamma}\,\widehat{\setminus}\,\widehat{z}(\langle z{\cdot}\beta\rangle\,\widehat{\beta}\;[v]\;\widehat{w}(\widehat{y}\langle z{\cdot}\delta\rangle\,\widehat{\delta}{\cdot}\epsilon)) & \rightarrow & (\setminus cut),(\setminus d),(\setminus d) \\
((P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\beta\rangle)\,\widehat{\beta}\;[v]\;\widehat{w}(\widehat{y}((P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\delta\rangle)\,\widehat{\delta}{\cdot}\epsilon) & \rightarrow & (a\not\setminus),(gc\not\setminus),(\setminus a),(\setminus gc) \\
(P\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\beta\rangle)\,\widehat{\beta}\;[v]\;\widehat{w}(\widehat{y}(Q\widehat{\gamma} \dagger \widehat{z}\langle z{\cdot}\delta\rangle)\,\widehat{\delta}{\cdot}\epsilon) & &
\end{array}$$

We cannot simulate this. Instead, we get



Of course, the shared $\mathsf{cut}(P,\alpha,x,Q)$ can be reduced only *once*.

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.

3 Dealing with α-conversion

In this section we will discuss a number of solutions to the problem of α-conversion, a well-known implementation issue. Example A.1 (see the appendix) shows that α-conversion is needed to some extent in any implementation of this calculus. In the λ-calculus, to correctly reduce a term like $(\lambda xy.xy)(\lambda xy.xy)$, α-conversion is essential. Without it, one would get

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

In this section, we will define 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. There the only reduction rule is $(\lambda x.M)N \rightarrow M[N/x]$, where substitution is supposed to be *immediate*. Since substitution is instantaneous and invisible, in the sense that we cannot foresee *where* in M the term N will be inserted, it is impossible to detect α-conflicts without having any further information available. In particular, it is impossible to predict (or even prevent) α-conflicts that typically depend on bindings occurring *inside* M and N, without significantly extending the reduction system.

To consider substitution as a separate syntactic structure implies moving from the λ-calculus to $\lambda\mathbf{x}$, and is of course a natural step when moving towards an implementation. For $\lambda\mathbf{x}$ the situation is slightly better, in that now substitution is an explicit part of term manipulation, so the chance of being able to deal with the creation of an α-conflict while reducing is higher. In fact, α-conflicts can be avoided by changing the rule

$$(\lambda y.M)\langle x{:=}N\rangle \rightarrow \lambda y.(M\langle x{:=}N\rangle),$$

into

$$(\lambda y.M)\langle x:=N\rangle \to \lambda z.(M\langle y:=z\rangle\langle x:=N\rangle),$$

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

In $\mathcal{X}$, the solution we will present is similar, but better. Since we can use side-conditions on our rules, we can say:

$$(\lambda y.M)\langle x:=N\rangle \to \lambda z.(M\langle y:=z\rangle\langle x:=N\rangle), \textit{ if } y \textit{ bound in } N$$

(notice that we can assume that y does not occur free in N). Using this approach in our solution, not only is the α-conflict solved, but also detected, all within the reduction system of $\mathcal{X}$ itself.

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, that states that an identifier should not appear both free and bound in a context, and normally the statement 'replacing bound identifiers to avoid clashes' is used, thus side-stepping the problem.

We will try to deal with the problem more formally. To this purpose, we will introduce a notion of α-safety. (The solution proposed in [5] was to avoid the sharing of graphs that were involved in multiple cuts with other graphs.)

Definition 3.1 (α-SAFETY) We call a circuit ($\mathcal{X}$-graph) *α-safe* if 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 preserves 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.

3.1 Preserving α-safety

We will set up our rewrite system in such a way that it avoids α-conversion conflicts; observe that such a conflict is introduced in the following step (see Example A.1):

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

Notice that μ is both bound and free in $(\widehat{y}\langle y\cdot\mu\rangle\,\widehat{\mu}\cdot\delta)\,\widehat{\delta}\dagger\,\widehat{y}\langle y\cdot\mu\rangle$. It is easy to see that rule (*exp-imp*) is *not* α-safe; for the left-hand side $(\widehat{y}R\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\,\widehat{k}(Q\widehat{\delta}[k]\,\widehat{w}P)$ to be α-safe, the connectors y and w, and μ and δ are allowed to be the same: they are not nested. However, this is no longer true for the right-hand sides: μ and δ occur nested in the first alternative, and y and w in the second, which might force an α-conversion to be necessary. Also, there is no reason to assume that δ or w do not appear bound in R.

Notice that we can always accomplish α-conversion, or renaming of bound connectors, for these particular clashes by the explicit renaming feature of $\mathcal{X}$, using new cuts such as $\langle v\cdot\delta\rangle\,\widehat{\delta}\,\diagdown\,\widehat{y}P$ or $P\,\widehat{\beta}\,\diagup\,\widehat{v}\langle v\cdot\delta\rangle$ to rename y by v, or β by δ respectively in P, where v,δ are fresh (see Lemma 1.7).

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

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

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:

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

Likewise, there are two alternatives for the second choice of rule *Iexp-imp*):

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

We need to do this for each rule where a possible α-conflict is introduced, like $(\nwarrow cut)$:

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

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). Almost all (exceptions are $(d\nearrow)$, $(cap\nearrow)$, $(\nwarrow d)$, and $(\nwarrow cap)$) propagation rules should be treated; of the logical rules, only rule $(exp\text{-}imp)$ needs dealing with as specified above. The advantage of this approach is that α-conversion itself is detected and dealt with. The computational cost is low compared to the approach defined in [5]; the price to pay is an increase in the number of rules.

However, it is straightforward to show that, modified this way, the notion of rewriting on $\mathcal{X}$ is α-safe. Let $P \rightarrow_\alpha Q$ stand for the notion of rewriting on $\mathcal{X}$ obtained by changing the rules as suggested above.

Theorem 3.2 *If P is α-safe, and $P \rightarrow_\alpha Q$, then Q is α-safe.*

3.2 Optimising the solution: garbage collection

A first optimisation is to check whether the renaming is actually necessary, by checking if the connector to rename occurs free in the graph. To that purpose, justified by Lemma 1.7, we replace rules $(cap\nearrow)$ and $(\nwarrow cap)$ by the (garbage collection) rules

$$\begin{array}{rcl}Q\widehat{\delta}\nwarrow\widehat{x}P & \rightarrow & P \quad \textit{if } x \notin fs(P), P \textit{ pure} \\ P\widehat{\alpha}\nearrow\widehat{v}Q & \rightarrow & P \quad \textit{if } \alpha \notin fs(P), P \textit{ pure}\end{array}$$

This has as positive side-effect that the renaming process does not traverse (and copy) all of the sub-graph, but just the part that contains the conflicting connector.

3.3 Optimising the solution: avoiding deactivation

A less obvious optimisation to the solution given above is based on the observation that the graphs that we now α-convert in the propagation rules, perhaps need not be changed at all. In particular, if we assume $P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R)$ to be α-safe, we can allow $\alpha = \beta$ (notice that $x \neq y$). Now applying rule $(\nwarrow cut)$ gives

$$P\widehat{\alpha}\nwarrow\widehat{x}(Q\widehat{\alpha}\dagger\widehat{y}R) \rightarrow (P\widehat{\alpha}\nwarrow\widehat{x}Q)\widehat{\alpha}\dagger\widehat{y}(P\widehat{\alpha}\nwarrow\widehat{x}R)$$

This is, by definition, an unsafe reduction, since we have created two nested binders for α. However, the *active* cuts $P\widehat{\alpha}\nwarrow\widehat{x}Q$ and $P\widehat{\alpha}\nwarrow\widehat{x}R$ will connect all α in P and x in Q and R,

the effect of which will be a term in which all the original α in P have disappeared, and only those in Q will remain. In contrast, the α bound in the *inactive* (outermost) cut are those occurring in Q. We can activate the outermost cut, but it will be impossible to have it propagate over the already active cuts, so it will be impossible to have the outermost cut involve the α in P.

The only caution we need to take is when the propagation meets a capsule (so we get a case of $P\widehat{\alpha}\backslash\!\!\backslash\,\widehat{x}\langle x{\cdot}\gamma\rangle$; notice that γ might be α). Now the cut gets deactivated via rule $(\backslash\!\!\backslash d)$, making it possible for the outermost cut to propagate over the innermost. In order to avoid this, the deactivation rules $(d\not\prec)$ and $(\backslash\!\!\backslash d)$ need to be removed. In order to not lose any results like normal forms, we need to add the rules

$$\begin{array}{rcll} \langle z{\cdot}\beta\rangle\,\widehat{\beta}\not\prec\widehat{x}\langle x{\cdot}\gamma\rangle & \to & \langle z{\cdot}\gamma\rangle & \\ \langle z{\cdot}\beta\rangle\,\widehat{\beta}\not\prec\widehat{x}(Q\widehat{\alpha}\,[x]\,\widehat{y}R) & \to & Q\widehat{\alpha}\,[z]\,\widehat{y}R & x \textit{ introduced in } Q\widehat{\alpha}\,[x]\,\widehat{y}R \\ \langle z{\cdot}\beta\rangle\,\widehat{\beta}\not\prec\widehat{x}P & \to & \langle z{\cdot}\beta\rangle\,\widehat{\beta}\backslash\!\!\backslash\,\widehat{x}P & x \textit{ not introduced in } P \end{array}$$

(and similar rules for $\backslash\!\!\backslash$) that act as short-cuts. Now we are certain that no conflict can occur since there is no chance that propagating cuts can 'swap scope'.

3.4 Optimising the solution: modifying rules $(cut\not\prec)$ and $(\backslash\!\!\backslash cut)$

In the optimisation above, only the possibility of the outermost cut propagating over the innermost creates a problem, since the wrong binding can then occur. Therefore, another solution would be to define rewriting on $\mathcal{X}$ without rules $(cut\not\prec)$ and $(\backslash\!\!\backslash cut)$. This could affect the reduction system, in that certain reduction sequences in full $\mathcal{X}$ need not have a counterpart; for example, reduction might not terminate.

Alternatively, we can add an extra side-condition to the rules:

$$\begin{array}{rcll} (\backslash\!\!\backslash cut'): & P\widehat{\alpha}\backslash\!\!\backslash\,\widehat{x}(Q\widehat{\beta}\dagger\widehat{y}R) & \to & (P\widehat{\alpha}\backslash\!\!\backslash\,\widehat{x}Q)\widehat{\beta}\dagger\widehat{y}(P\widehat{\alpha}\backslash\!\!\backslash\,\widehat{x}R) \quad x\neq y \\ (cut\not\prec'): & (Q\widehat{\beta}\dagger\widehat{y}R)\widehat{\alpha}\not\prec\widehat{x}P & \to & (Q\widehat{\alpha}\not\prec\widehat{x}P)\widehat{\beta}\dagger\widehat{y}(R\widehat{\alpha}\not\prec\widehat{x}P) \quad \alpha\neq\beta \end{array}$$

This will avoid the conflict by blocking a reduction, but only if a clash would occur. It limits the applicability of rules $(cut\not\prec)$ and $(\backslash\!\!\backslash cut)$ by not allowing their application when needed, but without eliminating them altogether.

4 Typing for $\mathcal{X}$

The notion of type assignment on $\mathcal{X}$ that we present in this section is the basic implicative system for Classical Logic.

Definition 4.1 (Types and Contexts) *i*) The set of types $\mathcal{T}_C$, ranged over by A, B, is defined over a set of *type-variables* $\mathcal{V}=\{\varphi_1,\varphi_2,\varphi_3,\ldots\}$ by the grammar: $A,B ::= \varphi \mid A{\to}B$. The types considered in this paper are normally known as *Curry* types.

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 defined by:

$$\begin{array}{rcll} \Gamma, x{:}A & = & \Gamma\cup\{x{:}A\}, & \text{if } \Gamma \text{ is not defined on } x \\ & = & \Gamma, & \text{if } x{:}A\in\Gamma \end{array}$$

Therefore, when writing a context as $\Gamma, x{:}A$, this implies that $x{:}A\in\Gamma$, or Γ is not defined on x. We write $\Gamma\backslash x$ for the context from which the statement concerning x, if any, has been removed. *Contexts of plugs* Δ, and the notations $\alpha{:}A,\Delta$ and $\Delta\backslash\alpha$ are defined in a similar way.

Definition 4.2 (Typing for $\mathcal{X}$) *i) Type judgements* are expressed via a ternary relation $P : \Gamma \vdash \Delta$, where Γ is a context of *sockets* and Δ is a context of *plugs*, and P is a circuit. We say that P is the *witness* of this judgement.

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

$$(cap): \frac{}{\langle y{\cdot}\alpha\rangle : \Gamma, y{:}A \vdash \alpha{:}A, \Delta} \qquad (imp): \frac{P : \Gamma \vdash \alpha{:}A, \Delta \quad Q : \Gamma, x{:}B \vdash \Delta}{P\widehat{\alpha}\,[y]\,\widehat{x}Q : \Gamma, y{:}A{\to}B \vdash \Delta}$$

$$(exp): \frac{P : \Gamma, x{:}A \vdash \alpha{:}B, \Delta}{\widehat{x}P\widehat{\alpha}{\cdot}\beta : \Gamma \vdash \beta{:}A{\to}B, \Delta} \qquad (cut): \frac{P : \Gamma \vdash \alpha{:}A, \Delta \quad Q : \Gamma, x{:}A \vdash \Delta}{P\widehat{\alpha} \dagger \widehat{x}Q : \Gamma \vdash \Delta}$$

We write $P : \Gamma \vdash \Delta$ if there exists a derivation that has this judgement in the bottom line.

Notice that, in $P : \Gamma \vdash \Delta$, the circuit P acts as a *witness* of the judgement; Γ and Δ carry the types of the free connectors in P, as unordered sets. There is no notion of type for P itself, instead the derivable statement shows how P is connectable. In fact, this notion of type assignment on $\mathcal{X}$-circuits is strikingly similar to that on *processes* of the π-calculus. The relation between $\mathcal{X}$ and π, and the implication of that relation on the connection between Classical Logic and π, is the subject of ongoing research.

We have the following result:

Theorem 4.3 (Witness reduction [4]) *If $P : \Gamma \vdash \Delta$, and $P \to Q$, then $Q : \Gamma \vdash \Delta$.*

We will now define a notion of *principal contexts* by providing an algorithm *pC* that, given a circuit P, returns a pair of contexts (Γ, Δ); as is common with Curry types, this will use the operation of substitution. Principal contexts for circuits are defined using Robinson's unification algorithm [14]. We will show that P is then a witness for $\Gamma \vdash \Delta$, and show that this pair is indeed the most general.

Definition 4.1 (Principal contexts) The procedure $pC :: \mathcal{X} \to \langle \Gamma; \Delta \rangle$ is defined in Figure 2, where *typeof* $\mathtt{c}\ \Psi$ (with $\mathtt{c}$ a connector, and Ψ a context) returns A, if $x{:}A \in \Psi$, else *fresh*, i.e. a unused type-variable, and *unifyCont* $\Psi_1\ \Psi_2$ returns a minimal substitution that, when applied to Ψ_1 and Ψ_2, generates two contexts that have the same type for each connector that occurs in both.

Notice that *pC* does not return a type for P, but a consistent way of assigning types to its free connectors.

We have the following result:

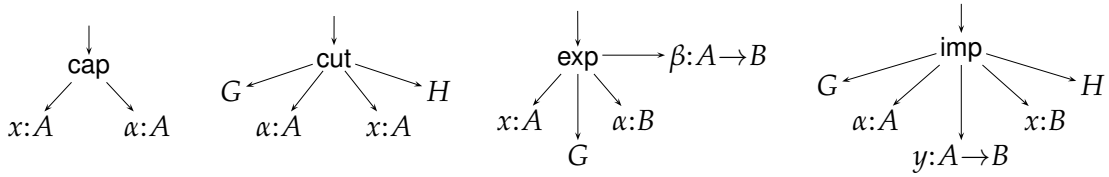
Theorem 4.2 (Soundness and Completeness of *pC*)

i) Soundness: If $pC\ P = (\Gamma, \Delta)$, then $P : \Gamma \vdash \Delta$.

ii) Completeness: If $P : \Gamma \vdash \Delta$, then there exist Γ_p and Δ_p, and a substitution S such that $pC\ P = (\Gamma_p, \Delta_p)$, and $S\Gamma_p \subseteq \Gamma$ and $S\Delta_p \subseteq \Delta$.

The notion of type assignment on $\mathcal{X}$-circuits extends easily to their term graph interpretation.

Definition 4.3 (Type assignment on $\mathcal{X}$-graphs) Type assignment on $\mathcal{X}$-graphs is defined as a (partial) labelling of graphs with types, using the following local constraints:



$$
\begin{array}{rcll}
pC\,\langle x{\cdot}\alpha\rangle & = & \langle x{:}\psi;\alpha{:}\psi\rangle & \\
\quad \textit{where } \psi & = & \textit{fresh} & \\
pC\,\widehat{x}P\widehat{\alpha}{\cdot}\beta & = & \langle \Gamma\backslash x;(\Delta\backslash\alpha)\cup\beta{:}A{\to}B\rangle, & \textit{if } \beta\notin\Delta \\
 & = & S\langle\Gamma\backslash x;\Delta\backslash\alpha\rangle, & \textit{if } \beta{:}C\in\Delta \\
\quad \textit{where } \langle\Gamma;\Delta\rangle & = & pC\,P & \\
A & = & \textit{typeof } x\ \Gamma & \\
B & = & \textit{typeof } \alpha\ \Delta & \\
S & = & \textit{unify } C\ (A{\to}B) & \\
pC\,P\widehat{\alpha}\,[y]\,\widehat{x}Q & = & S_2{\circ}S_1\langle\Gamma_P\cup(\Gamma_Q\backslash x)\cup y{:}A{\to}B;(\Delta_P\backslash\alpha)\cup\Delta_Q\rangle, & \textit{if } y\notin S_2{\circ}S_1\,\Gamma_P\cup(\Gamma_Q\backslash x) \\
 & = & S_3{\circ}S_2{\circ}S_1\langle\Gamma_P\cup(\Gamma_Q\backslash x);(\Delta_P\backslash\alpha)\cup\Delta_Q\rangle, & \textit{if } y{:}C\in S_2{\circ}S_1\,\Gamma_P\cup(\Gamma_Q\backslash x) \\
\quad \textit{where } \langle\Gamma_P;\Delta_P\rangle & = & pC\,P & \\
\langle\Gamma_Q;\Delta_Q\rangle & = & pC\,Q & \\
A & = & \textit{typeof } \alpha\ \Delta_P & \\
B & = & \textit{typeof } x\ \Gamma_Q & \\
S_1 & = & \textit{unifyCont } \Gamma_P\ ((\Gamma_Q\backslash x)) & \\
S_2 & = & \textit{unifyCont } (S_1\Delta_P\backslash\alpha)\ (S_1\Delta_Q) & \\
S_3 & = & \textit{unify } C\ (S_2{\circ}S_1\,A{\to}B) & \\
pC\,P\widehat{\alpha}\,\dagger\,\widehat{x}Q & = & S_3{\circ}S_2{\circ}S_1\langle\Gamma_P\cup\Gamma_Q\backslash x;(\Delta_P\backslash\alpha)\cup\Delta_Q\rangle & \\
\quad \textit{where } \langle\Gamma_P;\Delta_P\rangle & = & pC\,P & \\
\langle\Gamma_Q;\Delta_Q\rangle & = & pC\,Q & \\
A & = & \textit{typeof } \alpha\ \Delta_P & \\
B & = & \textit{typeof } x\ \Gamma_Q & \\
S_1 & = & \textit{unifyCont } \Gamma_P\ (\Gamma_Q\backslash x) & \\
S_2 & = & \textit{unifyCont } (S_1\Delta_P\backslash\alpha)\ (S_1\Delta_Q) & \\
S_3 & = & \textit{unify } (S_2{\circ}S_1\,A)\ (S_2{\circ}S_1\,B) &
\end{array}
$$

Figure 2: The principal contexts algorithm

Notice that only leaves get assigned types.

We write $G : \Gamma \vdash \Delta$ if there exists a type assignment on G such that

$$\{x{:}A \mid x \textit{ occurs free in } G\} \ \subseteq\ \Gamma \textit{ and } \{\alpha{:}A \mid \alpha \textit{ occurs free in } G\} \ \subseteq\ \Delta.$$

Notice that a leaf like x occurs exactly *once* in an $\mathcal{X}$-graph, so x can be typed only once, so Γ and Δ are indeed contexts.

This notion is non-standard in that *only* the leaves of a well-typed graph are decorated with types. This notion therefore differs from any notion of type assignment defined on term (graph) rewriting systems in the past. Also, since all the connectors mentioned in the rules can appear in the sub-graphs (like x and α in G in the case for exp), these constraints then actually state that also 'within' G these connectors carry the same type: they appear, after all, only *once* in the graph.

We can now show the following property, that relates type assignment for circuits to type assignment for (initial) $\mathcal{X}$-term graphs. Remember that we have the restriction that, by alpha conversion, all bound connectors in P are assumed to be different.

Lemma 4.4 *i)* $P : \Gamma \vdash \Delta$ *if and only if* $[\![P]\!] : \Gamma \vdash \Delta$.

ii) If $G : \Gamma \vdash \Delta$, *then* $Unrv\ G : \Gamma \vdash \Delta$.

The converse of part (*ii*) does not hold, since a bound connector that occurs only once in G can occur twice in $Unrv\ G$, where it can be typed with different types.

The notion of type assignment on term graphs also induces a notion of type assignment on the rewriting rules. The natural notion of preservation of type assignment under reduction of $\mathcal{X}$-graphs is formulated as usual, and can be achieved as a consequence of Theorem 4.3, observing that the sharing of nodes which would otherwise be copied causes no conflict on the type assignment.

Lemma 4.5 If $G : \Gamma \vdash \Delta$ and $G \to_{\mathrm{G}} G'$, then $G' : \Gamma \vdash \Delta$.

It is also possible to achieve this result by showing that the notion of type assignment on term graphs presented here satisfies the criterion on typeable rewrite rules of [6, 1, 3]; this states that for rewriting to preserve types, it is necessary that for each rewrite rule the principal pair for the left-hand side should be a viable pair for the right-hand side. We now come to the main result of this section.

Lemma 4.6 If $P : \Gamma \vdash \Delta$ and $[\![P]\!] \to_{\mathrm{G}} G$, then there exists Q such that $P \to Q$, $[\![Q]\!] = Unrv\ G$, and $G : \Gamma \vdash \Delta$. Also, if $G : \Gamma' \vdash \Delta'$, then $Q : \Gamma' \vdash \Delta'$.

5 Conclusions and future work

We feel that the solution for α-conversion as presented in this paper is far from optimal. In fact, the implementation of the tool has brought several better solutions to light that, as for the optimisations discussed here, would not preserve α-safety, but would define a notion of reduction that only renames when really necessary. The formal definitions and verifications will be subject of future work.

We aim to extend the notion of principal context to graphs, and study their relationship with principal contexts for circuits.

References

[1] S. van Bakel. Rank 2 Intersection Type Assignment in Term Rewriting Systems. *Fundamenta Informaticae*, 2(26):141–166, 1996.

[2] S. van Bakel. Intersection and Union Types for $\mathcal{X}$. In *ITRS '04*, ENTCS, 2004.

[3] S. van Bakel and M. Fernández. Normalization Results for Typeable Rewrite Systems. *Information and Computation*, 2(133):73–116, 1997.

[4] S. van Bakel, S. Lengrand, and P. Lescanne. The language $\mathcal{X}$: circuits, computations and classical logic. *Submitted*, 2005.

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

[6] S. van Bakel, S. Smetsers, and S. Brock. Partial Type Assignment in Left Linear Applicative Term Rewriting Systems. In *CAAP '92*, LNCS 581, 300–321, 1992.

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

[8] 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.

[9] 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.

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

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

[12] G. Gentzen. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift*, 39:176–210 and 405–431, 1935. English translation in [16], pages 68–131.

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

[14] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

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

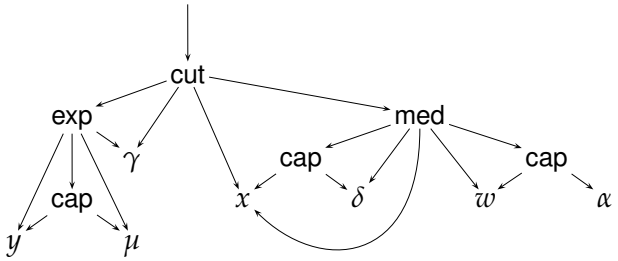
[16] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1969.

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

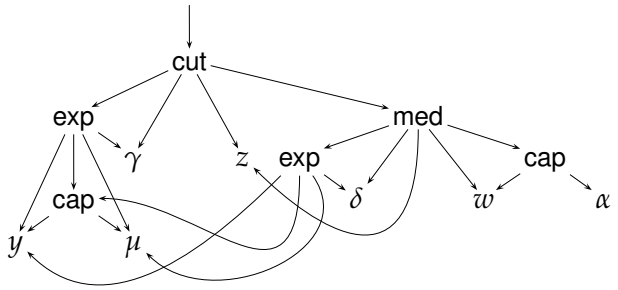
Appendix A Alpha collision example

We will illustrate both the need to deal with α-conversion as well as the working of the graph rewriting engine with its sharing.

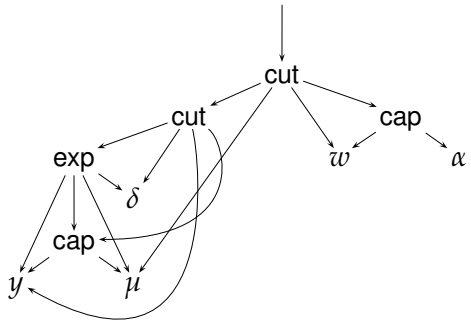
Example A.1 Take the following circuit (graph): $(\widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\,\widehat{x}(\langle x{\cdot}\delta\rangle\,\widehat{\delta}\,[x]\;\widehat{w}\langle w{\cdot}\alpha\rangle)$



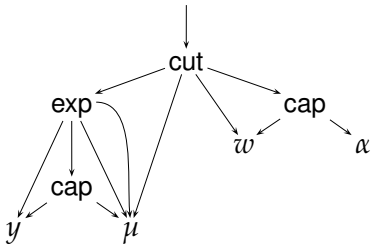
Applying the rewrite rules $(\curlywedge a)$, $(\curlywedge \mathit{imp\text{-}out})$, $(\curlywedge \mathit{cap})$, $(\curlywedge d)$, and (exp) will generate the circuit $(\widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}\cdot\gamma)\,\widehat{\gamma}\dagger\,\widehat{z}((\widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}\cdot\delta)\,\widehat{\delta}\,[z]\;\widehat{w}\langle w{\cdot}\alpha\rangle)$.



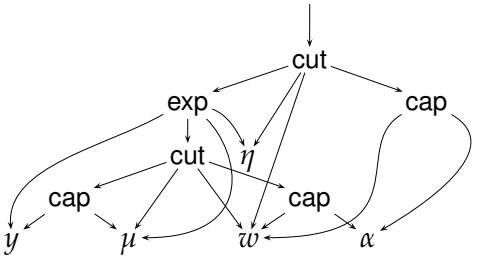
As is clear from this graph, the capsule on the left is now shared, and there are *two* binders to both y and μ, coming from both export terms. Continuing the execution (via rule $(\mathit{exp\text{-}imp})$) yields a graph that represents the circuit $((\widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}\cdot\delta)\,\widehat{\delta}\dagger\,\widehat{y}\langle y{\cdot}\mu\rangle)\,\widehat{\mu}\dagger\,\widehat{w}\langle w{\cdot}\alpha\rangle$.



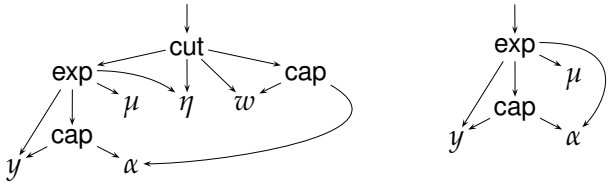
Notice that now we have an α-conversion conflict: there are two nested binders to μ. Even if we assume that the innermost binds the strongest, so that the left free μ is bound by the innermost, and the right free μ by the outermost, this only solves the conflict here. If we now first reduce the inner cut using (exp), we obtain: $(\widehat{y}\langle y{\cdot}\mu\rangle\,\widehat{\mu}\cdot\mu)\,\widehat{\mu}\dagger\,\widehat{w}\langle w{\cdot}\alpha\rangle$.



Notice that the right-most free μ is bound by the outermost binder, but the rewrite rules will generate the wrong circuit. To reduce the cut in this circuit, we first check if μ is introduced; this is not so, since $\mu \in fp(\langle y \cdot \mu \rangle)$. So we need to propagate the cut, and obtain, using rules $(a\not\curvearrowright)$, $(exp\text{-}out\not\curvearrowright)$ and $(d\not\curvearrowright)$, $(\widehat{y}(\langle y \cdot \mu \rangle \widehat{\mu} \dagger \widehat{w}\langle w \cdot \alpha \rangle)\widehat{\mu} \cdot \eta)\widehat{\eta} \dagger \widehat{w}\langle w \cdot \alpha \rangle$



which reduces by rules (cap) to $(\widehat{y}\langle y \cdot \alpha \rangle \widehat{\mu} \cdot \eta)\widehat{\eta} \dagger \widehat{w}\langle w \cdot \alpha \rangle$ and then by (exp) to $\widehat{y}\langle y \cdot \alpha \rangle \widehat{\mu} \cdot \alpha$.



This is not the correct result, since

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

where μ is introduced, we should have obtained $\widehat{y}\langle y \cdot \mu \rangle \widehat{\mu} \cdot \alpha$ via (exp).

Applying the solution of Section 3.1, we have, instead of the problematic step

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

the correction

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