# Explicit substitution for graphs

Vincent van Oostrom
Universiteit Utrecht
Faculteit Wijsbegeerte, Theoretische Filosofie
Heidelberglaan 8, 3584 CS Utrecht, The Netherlands
Vincent.vanOostrom@phil.uu.nl

## Abstract

*We present an atomic decomposition of substitution into erasure, duplication and (for bound variables) scoping.*

## 1. Introduction

Substitution pervades logic. Implementing logic one immediately realises that substitution is not an atomic operation. Thus one is faced with the question how to implement substitution. In the seminal work by De Bruijn on Automath this question was answered by introducing what is now known as an explicit substitution calculus on terms. Here, working on graphs instead of on terms, we present explicit substitution for graphs. In particular, we show how substitution can be made explicit by means of three atomic operators for erasure, duplication, and scoping. Here we aim to present the basic ideas in an intuitive way. We illustrate the issues for rewriting systems, although similar ideas can be found in many other sub-fields of logic.

## 2. Linear substitution

A term rewriting system (TRS) is given by an alphabet together with a set of rewrite rules over the alphabet [9]. As an example consider the TRS $\mathcal{A}$ for addition of (unary) natural numbers having rules:

$$\begin{array}{rcl} x+0 & \to & x \\ x+S(y) & \to & S(x+y) \end{array}$$

Using these rules we may find the reduction $\mathcal{R}$ given by

$$S(S(0)+S(0)) \to_{\mathcal{A}} S(S(S(0)+0))) \to_{\mathcal{A}} S(S(S(0)))$$

For instance, the first step is obtained by observing that the underlined sub-term of $S(\underline{S(0)+S(0)})$ is a substitution instance of the left-hand side $x+S(y)$ of the second rule (substitute $S(0)$ for $x$ and $0$ for $y$). The step is obtained by replacing this sub-term by taking the same substitution instance for the right-hand side of the same rule, yielding $S(\overline{S(S(0)+0)})$, where the replaced sub-term is over-lined.

The reason for being so detailed about this here, is that we want to stress that rewriting is a three-phase process consisting of *matching* (decomposing a term into a context and a left-hand side), *replacement* (replacing the left-hand side by the corresponding right-hand side), and *substitution* (composing the context and the right-hand side into a term). Whereas usually emphasis is put on the second phase, here we will be interested in the third phase, substitution.

In the case of addition, the substitution phase is always simple since each rule of $\mathcal{A}$ is *linear*. That is, every variable occurs exactly once in both its sides (or not at all). For this reason, substitution can be thought of mathematically as a permutation. Implementing terms by graphs, and term rewrite rules by graph rewrite rules, so-called term graph rewriting [8]. permutation boils down to rewiring which can be performed in constant time. In Figure 1 the graph rewrite

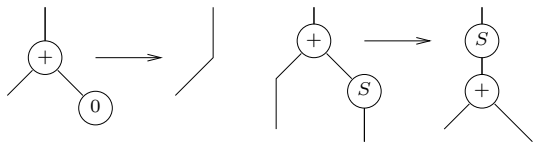


**Figure 1. Graph rewrite rules for $\mathcal{A}$**

system (GRS) corresponding to the TRS $\mathcal{A}$ is presented and Figure 2 displays the graph reduction corresponding to the reduction $\mathcal{R}$. Note that also a graph rewrite step can be decomposed into the three phases mentioned above. Typical other examples of linear term rewrite rules are the rules for commutativity and associativity:

$$\begin{array}{rcl} x+y & \to & y+x \\ (x+y)+z & \to & x+(y+z) \end{array}$$

Unfortunately, not all term rewrite rules are linear.

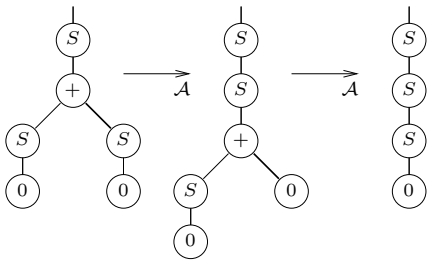

**Figure 2. Graph reduction for $\mathcal{R}$**

## 3. Non-linear substitution

Consider the extension $\mathcal{M}$ of the TRS $\mathcal{A}$ for addition, by the rules for multiplication:

$$\begin{array}{rcl} x \times 0 & \to & 0 \\ x \times S(y) & \to & x + (x \times y) \end{array}$$

with typical reduction $\mathcal{S}$ given by

$$S(0) \times S(0) \to_{\mathcal{M}} S(0) + (S(0) \times 0) \to_{\mathcal{M}} S(0) + 0$$

Note that neither of the rules for multiplication is linear. The first is *erasing*: the variable $x$ appears in its left-hand side, but not in its right-hand side. The second is *duplicating*: the variable $x$ appears once in its left-hand side, but twice in its right-hand side. To represent replication we introduce the eraser node $\circledcirc$ and the duplicator node $\triangledown$ in the graph representation of these rules in Figure 3. The formal rewrite

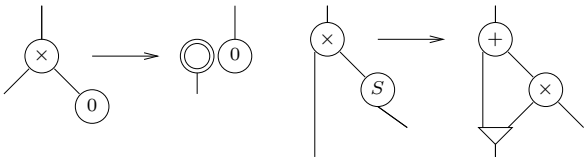

**Figure 3. Graph rewrite rules for $\mathcal{M}$**

rules for these replicator nodes will be presented in the next section, but the idea should be clear already from looking at the implementation of the reduction $\mathcal{S}$ in Figure 4: the argument connected to such a node is node-wise replicated the appropriate (0 or 2) number of times. The important point is that the substitution phase of reduction has become non-trivial; replication takes time linear in the size of the replicated argument. Indeed, substitution steps, indicated by the subscript $x$, form the majority of the steps in Figure 4. The idea is then to delay such steps.

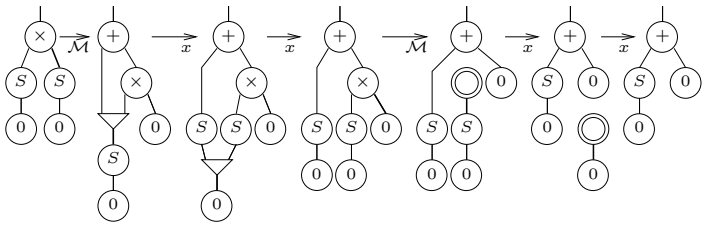

**Figure 4. Graph reduction for $\mathcal{S}$**

## 4. Explicit substitution rules

The graph rewrite rules for the eraser and duplicator are both instances of the two rule schemata in Figure 5. In a

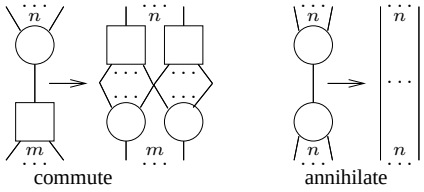


**Figure 5. Schemata for explicit substitution**

slogan: distinct symbols commute; identical symbols annihilate. All our rules for substitution operators will be instances of only these two simple schemata. In fact, for the moment just commutation suffices. In Figure 6 commutation is spelled out between on the one hand the replicators $\circledcirc$ and $\triangledown$ and on the other hand the function symbols $0$ and $S$. Note that the right-hand side of the commutation rule for

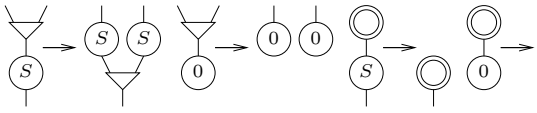

**Figure 6. Commutation of $\triangledown$, $\circledcirc$ with $S$, $0$**

$\circledcirc$ and $0$ in Figure 6 is the empty graph, as in this case both $n$ and $m$ in the commutation rule of Figure 5 are zero.

## 5. Delaying explicit substitution

In the naive graph rewriting implementation of $\mathcal{M}$ as in Figure 4, an $\mathcal{M}$-step is followed by a number of substitution steps until none is possible anymore, after which the next $\mathcal{M}$-step takes place, etc.. Having an explicit representation of substitution gives one the freedom to break this pattern. For instance, it is intuitively clear that duplicating a subterm for which it takes an expensive computation to yield a

simple result is wasteful; computing the result first and then duplicating it saves half the time.

For terms, the delay of substitution is usually brought about by extending terms with the let-construct. For instance, applying a rule $x \times 2 \to x + x$ to $E \times 2$, yields $\mathsf{let}\, x = E \,\mathsf{in}\, x + x$ instead of $E + E$, which is a good thing when $E$ is expensive to compute. Hence, the let-construct can be viewed as an explicit substitution operator for terms.

For graphs, it suffices to break the pattern of reducing to substitution normal form after each rewrite step. Whereas up till now ordinary rewrite rules were only applied to graphs which were in fact trees, breaking the pattern leads to their application to graphs which are not trees.

**Remark 1.** In a *maximal* sharing discipline, as implemented in the ATerm library [3], *all* identical sub-terms are shared. That is, duplication is not just delayed but performed in the reverse direction to obtain maximal sharing.

## 6. Implementation

Stopping short of reaching the substitution normal form gives rise to the following adequacy questions, cf. [8]:

- Can one characterise the graphs representing terms?
- How do graph and term rewriting relate?

To give somewhat precise answers to these questions, it is useful to introduce a bit of notation. Let $\mathfrak{G}(t)$ denote the directed graph (in fact, tree) corresponding to the term $t$, obtained by directing all edges downward. Letting $\mathfrak{T}(G)$ denote the (unique if any) term $t$ such that $\mathfrak{G}(t)$ is the substitution normal form of $G$, we have that $\mathfrak{T} \circ \mathfrak{G}$ is the identity on terms. The standard answer to the first question then is: representing graphs are directed and acyclic (dags). An equivalent characterisation in terms of substitution is:

graphs whose substitution normal form is a finite tree.

Uniqueness of substitution normal forms follows from confluence which holds since the substitution rules are orthogonal to one another; they constitute an interaction net [6]. A first answer to the second question is:

**Lemma 2.** ***(commutation)*** *If $G \to H$, then $\mathfrak{T}(G) \multimap\!\!\to \mathfrak{T}(H)$, where $\multimap\!\!\to$ denotes multi-step reduction, contracting a number of redexes in a term simultaneously.*

***(progress)*** *If $\mathfrak{T}(G) \to s$, then there exist $G'$, $H$ such that $G \twoheadrightarrow_x G' \to H$, with the corresponding multi-step $\mathfrak{T}(G') \multimap\!\!\to \mathfrak{T}(H)$ contracting at least the redex contracted by $\mathfrak{T}(G) \to s$ (note that $\mathfrak{T}(G') = \mathfrak{T}(G)$).*

**Remark 3.** Depending on one's needs more stringent conditions can be put on the relationship, e.g. that graph rewriting of $\mathfrak{G}(t)$ should terminate if term rewriting of $t$ does so.

## 7. Cyclic substitution

Dropping the finiteness condition in the characterisation of representing graphs above allows for the implementation of (potentially) infinite terms. To see this, consider ones, the infinite streams of 1s

$$1{:}1{:}1{:}1{:}\ldots$$

For terms, this infinite stream can be brought about by means of the letrec-construct

$$\mathsf{letrec}\, x = 1{:}x \,\mathsf{in}\, x$$

For graphs, it suffices to forget the finiteness condition on their substitution normal form in the characterisation above, as then the stream can be represented as the graph on the left in Figure 7. Indeed, computing the substitution

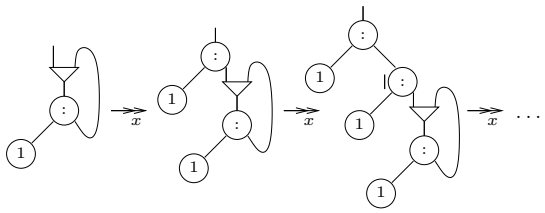


**Figure 7. Infinite substitution normal form**

normal form of this graph yields 'in the limit' an infinite tree representing the infinite stream of 1s as suggested in the figure. The results of the previous section for acyclic substitutions should extend to this cyclic case. We expect the implementation Lemma 2 can be shown by extending the theory of infinitary rewriting [9] from terms to graphs.

Note that we employ the same rule schemata for computing substitutions as before, and that combining them with an orthogonal TRS yields a combined system which is orthogonal, hence confluent. This is a bit surprising as it is well-known that collapsing on the one hand all the $g$s and on the other hand all the $f$s, in the infinite term $f(g(f(g(\ldots))))$ with respect to the orthogonal term rewrite rules $g(x) \to x$ and $f(x) \to x$, yields infinite terms $f(f(\ldots))$ and $g(g(\ldots))$ which are *not* joinable in the infinitary TRS. However, for their cyclic representation this is not a problem as shown in Figure 8; a so-called vicious circle [6] serves as the common reduct. A vicious circle is intuitively meaningless [9], but that's not needed here.

**Remark 4.** All infinite terms above represented by finite graphs are regular. Allowing the context-free substitutions to be introduced in the next section, non-regular ones such as the stream of natural numbers can be represented too.

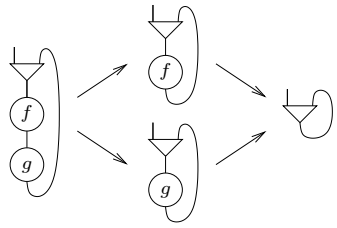


**Figure 8. Confluence using a vicious circle**

## 8 Scoping substitution

We now turn to implementing substitution for terms having binding symbols such as $\lambda$, $\forall$, $\sum$ etc. in an atomic manner. We treat the particular case of implementing $\beta$-reduction, i.e. substitution, for the $\lambda$-calculus [2]. As running example we take the Church-numeral $\underline{2}$ given by

$$\lambda x.\lambda y.x(xy)$$

First, we switch to the nameless $\lambda$-terms of [4], where each variable is replaced by a (unary) natural number indicating by which $\lambda$ above it in the syntax tree the variable was bound (counting from zero). The representation of $\underline{2}$ is

$$\lambda\lambda(\mathsf{S}0)((\mathsf{S}0)0)$$

Second, we reinterpret a successor $\mathsf{S}$ as an explicit scope operator [5]; in a slogan: $\mathsf{S}$ stands for scope. The idea is illustrated in Figure 9, displaying from left to right, the syntax

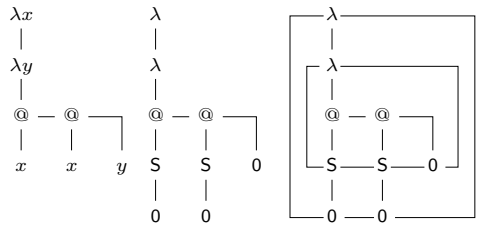


**Figure 9. Reinterpreting successor as scope**

tree of the ordinary $\lambda$-term $\underline{2}$, the syntax tree of its nameless version, and that tree again with scopes explicitly indicated by boxes. The boxes show that *binding* for an ordinary $\lambda$-term corresponds to *matching* for its nameless version: every $\mathsf{S}$ corresponds to a unique matching $\lambda$; any node below the $\mathsf{S}$ is *out of scope* of the $\lambda$, i.e. will not be affected by a substitution for its variable. Thus, $\lambda$-terms can be seen as *context-free* trees. The idea is to introduce the *scope* operator $\mathsf{S}$ into to our alphabet of substitution operators and implement substitution such that the matching structure is preserved. Unfortunately, this does not quite work because during $\beta$-reduction the neat nesting structure of boxes will be lost, they may partially overlap, and we must for each scope- and duplication-node individually keep track of how deep it is nested. To that end, we also index our operators as $\sqcup_i$ and $\triangledown_i$ for arbitrary depth $i$, setting $\triangledown = \triangledown_0$ and $\mathsf{S} = \sqcup_0$.

## 9. Translating $\lambda$-terms

We present an inductive translation of closed $\lambda$-terms into graphs built out of the explicit substitution operators and the function symbols $\lambda$ (*abstraction*) and @ (*application*). Here a term $t$ is *closed* if $0 \vdash t$ in the following inference system (to be read top–down)

$$\frac{}{\mathsf{S}i \vdash 0}\,0 \qquad \frac{\mathsf{S}i \vdash \mathsf{S}t}{i \vdash t}\,\mathsf{S} \qquad \frac{i \vdash \lambda t}{\mathsf{S}i \vdash t}\,\lambda \qquad \frac{i \vdash t_1 t_2}{i \vdash t_1 \quad i \vdash t_2}\,@$$

The nameless term $\underline{2}$ is closed, as shown in Figure 10. A

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0 \vdash \lambda\lambda(\mathsf{S}0)((\mathsf{S}0)0)}\,\lambda}{\mathsf{S}0 \vdash \lambda(\mathsf{S}0)((\mathsf{S}0)0)}\,\lambda}{\mathsf{SS}0 \vdash (\mathsf{S}0)((\mathsf{S}0)0)}\,@}{\dfrac{\dfrac{\mathsf{SS}0 \vdash \mathsf{S}0}{\mathsf{S}0 \vdash 0}\,\mathsf{S}}{\mathsf{S}0 \vdash 0}\,0 \qquad \dfrac{\mathsf{SS}0 \vdash (\mathsf{S}0)0}{\dfrac{\mathsf{SS}0 \vdash \mathsf{S}0}{\mathsf{S}0 \vdash 0}\,\mathsf{S} \qquad \mathsf{S}0 \vdash 0}\,@}\,0$$

**Figure 10. Derivation showing $\underline{2}$ is closed**

well-formed term $i \vdash t$ is mapped to a graph having $i + 1$ free ports, which is defined by induction and cases ($0$, $\mathsf{S}$, $\lambda$, and @) on the derivation, in Figure 11. Here a number $i$ next

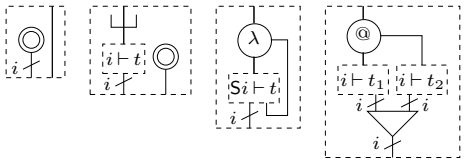


**Figure 11. From $\lambda$-terms to graphs.**

to a slashed edge represents that in fact the edge is a 'bus' consisting of $i$ edges (connected to an appropriate number of nodes). Figure 12 shows the application to $\underline{2}$.

## 10 Implementing $\beta$-reduction

$\beta$-reduction is implemented by the rule in Figure 13. As before substitution is dealt with by the two rule schemata of Figure 5 (now annihilate is needed). In addition, indices need to be updated, where an *update* is an increment of the index $i$ (if any) of a substitution operator, which takes place iff the other symbol is either $\lambda$ or $\sqcup_j$, with $i \geq j$.

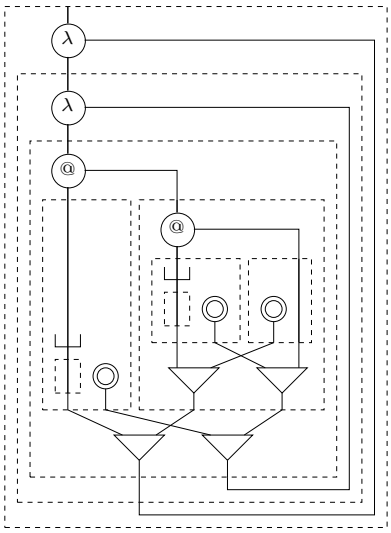

**Figure 12. The graph translation of $\underline{2}$.**

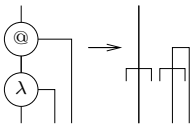

**Figure 13. Translation of $\beta$-reduction rule**

To get a flavour of this decomposition of $\beta$-reduction into atomic steps, in Figure 14 a part of the reduction of (an optimised translation of) $\underline{2}\,\underline{2}$ to graph normal form is shown. Although each of the individual steps is simple, it is easy to lose track of what is really happening, because there are so many steps. Recalling that application of Church-numerals is exponentiation, the final graph displayed should be a representation of the Church-numeral $\underline{4}$. Indeed it is. For an explanation as to why see [7]. There it is also shown that, as for the first-order case above, the implementation is adequate (Lemma 2). Again, the proof of adequacy does not depend on acyclicity, so should generalise to cyclic $\lambda$-terms.

## 11. Conclusion

We have presented an implementation of term rewriting based on keeping a clear distinction between on the one hand the implementation of substitution (the *substitution calculus* in the terminology of [10]), and on the other hand the implementation of operations on terms (here: the term rewrite rules). We have illustrated this for both the acyclic as well as the cyclic case, for first-order term rewriting and the $\lambda$-calculus. The atomic decomposition of substitution presented is simple (three operators), easy to implement (two rule schemata), versatile (both acyclic and cylic), and intuitive (erasure, duplication and scoping are found in many contexts). We conclude with mentioning two possible applications. Representing proofs: the graphs can be seen as atomic decompositions of (the box in) Girard's proof nets for multiplicative exponential linear logic. Implementing functional programming: the implementation of $\beta$-reduction is in fact optimal in the sense of [1]. It would be interesting to combine this with other techniques.

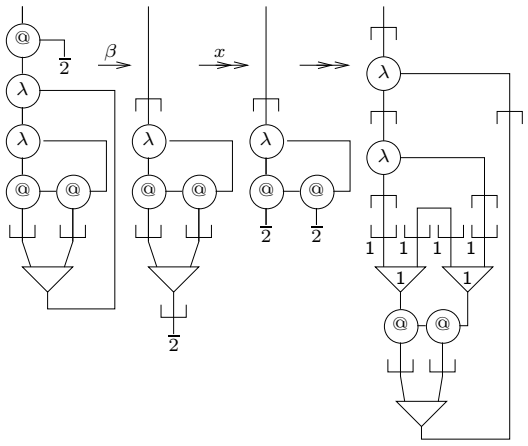

**Figure 14. Reduction of $\underline{2}\,\underline{2}$**

**Acknowledgements** I have benefited from discussions over the years with co-authors, colleagues and students on the topic of this note, for which I thank them.

## References

[1] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. CUP, 1998.
[2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
[3] M. v. d. Brand, H. de Jong, P. Klint, and P. Olivier. Efficient annotated terms. *Software – Practice & Experience*, 30:259–291, 2000.
[4] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indagationes Mathematicae*, 34:381–392, 1972.
[5] D. Hendriks and V. v. Oostrom. $\lambda$. In *CADE 19*, volume 2741 of *LNAI*, pages 136–150. Springer, 2003.
[6] Y. Lafont. Interaction nets. In *POPL 17*, pages 95–108. ACM Press, 1990.
[7] V. v. Oostrom, K.-J. v. d. Looij, and M. Zwitserlood. ]. Draft. Available from the first author's homepage, 2004.
[8] M. Sleep, M. Plasmeijer, and M. van Eekelen, editors. *Term Graph Rewriting*. John Wiley & Sons, 1993.
[9] Terese. *Term Rewriting Systems*. CUP, 2003.
[10] V. van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1994.