

A ‘new’ abstraction algorithm

M.A. Price

Department of Computer Science
The University of Bath
Bath
England

M.A.Price@bath.ac.uk

H. Simmons

Mathematical Foundations Group
The University
Manchester
England

H.Simmons@manchester.ac.uk

Abstract

We describe and illustrate an algorithm for simulating λ -abstraction which has been around for many years but seems to have been forgotten. Perhaps because the published version is wrong.

A λ -abstraction algorithm in a combinator calculus consumes a term A and an identifier x to return an x -free term $[x]A$ with reduction properties analogous to the λ -term $\lambda x . A$. Thus

$$([x]A)x \triangleright\triangleright A$$

where $\triangleright\triangleright$ is the combinator reduction relation. There are several such algorithms, some more efficient than others. In this context the names Schönfinkel, Curry, Feys, and Turner should be mentioned. A survey of these algorithms is given in [2].

On reading [4] we find it contains such an algorithm which seems to have been overlooked, or forgotten. The account is rather garbled and, in one respect, wrong, but perhaps the algorithm is worth remembering. In this note we describe the raw algorithm together with a more efficient version, and then use it to produce a new simulation of the S -combinator.

We work in a calculus which certainly has combinators

I K B C W

where

$$I p \triangleright\triangleright p \quad K q p \triangleright\triangleright q \quad B r q p \triangleright\triangleright r(qp) \quad C r q p \triangleright\triangleright r p q \quad W q p \triangleright\triangleright q p p$$

are their 1-step reductions. Here p, q, r are arbitrary terms. (In [4] the combinator W is called D , but here we use what is now the standard notation.) For what we do here it doesn't matter if these terms are primitive or compound. We also make reference to the combinator S where

$$S r q p \triangleright\triangleright (r p)(q p)$$

for arbitrary terms p, q, r . We will show how to simulate S using I, K, B, C, W . Of course, such simulations are known, but we produce a slightly ‘better’ one.

We write ∂A for the set of identifiers (variables) occurring free in the term A .

0.1 DEFINITION. (The raw G-algorithm) For each term A and identifier x the term $[x]A$ is generated by recursion over the structure of A using the clauses

$$\begin{aligned} (i) \quad [x]x &= I \\ (k) \quad [x]Z &= KZ && \text{if } x \notin \partial Z \\ (w) \quad [x](QP) &= W\left(\left(B(C([x]Q))\right)\left([x]P\right)\right) && \text{otherwise} \end{aligned}$$

for arbitrary terms P, Q, Z . (The different sizes of brackets helps with the parsing.) ■

It is routine to check that this does give an abstraction algorithm. Thus

$$\begin{aligned}
([x](QP))x &= \mathbf{W}\left(\left(\mathbf{B}(\mathbf{C}([x]Q))\right)([x]P)\right)x \\
&\triangleright \mathbf{B}(\mathbf{C}([x]Q))([x]P)xx \\
&\triangleright \mathbf{C}([x]Q)(([x]P)x)x \\
&\triangleright \left(\left([x]Q\right)x\right)\left(\left([x]P\right)x\right) \quad \triangleright QP
\end{aligned}$$

forms a basis of an inductive verification of the algorithm.

This raw G-algorithm is very like the algorithm devised independently by Schönfinkel and Curry. They used **I**, **K**, **S** and replaced (w) by

$$(s) \quad [x](QP) = \mathbf{S}([x]Q)([x]P)$$

for arbitrary terms P, Q .

Both these algorithms are rather inefficient in that they tend to produce long and unnecessary compounds. Both can be improved in the same way.

0.2 DEFINITION. (The cooked G-algorithm) For each term A and identifier x the term $[x]A$ is generated by recursion over the structure of A using the clauses

$$\begin{aligned}
(i) \quad [x]x &= \mathbf{I} \\
(k) \quad [x]Z &= \mathbf{K}Z && \text{if } x \notin \partial Z \\
(a) \quad [x](Qx) &= Q && \text{if } x \notin \partial Q \\
(b) \quad [x](QP) &= \mathbf{B}Q([x]P) && \text{if } x \notin \partial Q, x \in \partial P \\
(c) \quad [x](QP) &= \mathbf{C}([x]Q)P && \text{if } x \in \partial Q, x \notin \partial P \\
(w) \quad [x](QP) &= \mathbf{W}\left(\left(\mathbf{B}(\mathbf{C}([x]Q))\right)([x]P)\right) && \text{if } x \in \partial Q, x \in \partial P
\end{aligned}$$

for arbitrary terms P, Q, Z . Clause (a) takes precedence over clause (b). ■

If we modify this cooked algorithm by replacing (w) by (s), then we obtain the standard Curry-Feys algorithm.

The actual algorithm embedded in [4] is neither the raw nor the cooked version of the G-algorithm

0.3 OBSERVATION. (The half-baked G-algorithm) The actual algorithm in [4] is contained in the proofs of items 1.1 and 1.2. It uses the clauses (i, b, c, w) only. The exclusion of (a) makes this inefficient. More importantly, it is wrong. It does not use clause (k) and the combinator **K** can not be simulated using only **I**, **B**, **C**, **W**. ■

This G-algorithm may not be a delicacy, but it is not entirely without taste.

0.4 EXAMPLE. We produce a term S such that

$$Szyx \triangleright (zx)(yx)$$

for all identifiers x, y, z . We do this in three phases by calculating

$$Q = [x]\left(\left(zx\right)\left(yx\right)\right) \quad R = [y]Q \quad S = [z]R$$

using the cooked G-algorithm.

For the first phase we have

$$Q = [x]((zx)(yx)) = W\left(\left(B(C([x](zx)))\right)([x](yx))\right) = W\left(\left(B(Cz)y\right)\right)$$

by (w) and two uses of (a).

For the second phase we have

$$R = [y]Q = BW\left([y]\left(\left(B(Cz)y\right)\right)\right) = BW\left(\left(B(Cz)\right)\right)$$

using (b) and then (a).

Finally we have

$$S = [z]R = [z]\left(BW\left(\left(B(Cz)\right)\right)\right) = B(BW)\left([z]\left(B(Cz)\right)\right) = B(BW)\left(\left(BB\right)[z](Cz)\right)$$

to give

$$S = B(BW)(BBC)$$

by two uses of (b) and one of (a). ■

Of course, $\{B, C, W\}$ -simulations of S are not entirely unknown. Consider the three terms

$$S_f = B(B(BW)B)C \quad S_g = B(BW)(BBC) \quad S_h = B(B(BW)C)(BB)$$

where S_h appears in [1, 3], the term $S_g = S$ is generated above, and you will have to guess where S_f comes from. It can be checked that each of

$$S_f zyx \quad S_g zyx \quad S_h zyx$$

reduces to $(zx)(yx)$ in 6, 6, and 7 steps, respectively. Note also that both S_f and S_g are shorter than S_h . Can any of you better that?

Cheers.

We are grateful to Roger Hindley for some sobering comments.

References

- [1] H.P. Barendregt: *The lambda calculus its syntax and semantics*, North Holland (1988).
- [2] M.W. Bunder: Some improvements to Turner's algorithm for bracket abstraction, *J. Symbolic Logic*, 55 (1990) 656–669.
- [3] J.R. Hindley: *Basic simple type theory*, C.U.P (1997).
- [4] A. Grzegorzcyk: Recursive objects in all finite types, *Fund. Maths.* 54 (1964) 73–93.