Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 349 (2020) 25–48

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Criteria for Bracket Abstractions Design

Federico Flaviani[1]

Department of Computing and Information Technologies
Universidad Simón Bolívar
Caracas, Venezuela

Elias Tahhan Bitar[2]

Department of Pure and Applied Mathematics
Universidad Simón Bolívar
Caracas, Venezuela

Abstract

A bracket abstraction is a syntactic operator to abstract variables in combinatory logic. There are different algorithms for different combinator systems. In this paper we present a recursive algorithm scheme, which generates a family of brackets abstractions, in which all the bracket abstractions referenced here are found. In addition, theorems with certain hypotheses are enunciated about the scheme, which state that the resulting abstraction operators, has one property or another. Thus forming a criteria for designing bracket abstractions that comply with a given property.

Keywords: Bracket Abstraction; Recursive Scheme, Combinators, Injectivity.

1 Introduction

The combinatory logic can be understood as a rewriting system [1], on an application language with variables and constants. Application language is understood as a language $AL(Var, \Sigma_Z)$ with a set of variables Var and a signature $\Sigma_Z := Z \cup \{.\}$ consisting of a binary operator . and a set of constants Z. The description is recursively

- If $p \in Var$ or $p \in Z$ then $p \in AL(Var, \Sigma_Z)$,
- If $p, q \in AL(Var, \Sigma_Z)$ then $p.q \in AL(Var, \Sigma_Z)$.

To simplify notation, it is assumed that the association is made to the left and pq denotes $p.q$.

[1] Email: fflaviani@usb.ve
[2] Email: etahhan@usb.ve

https://doi.org/10.1016/j.entcs.2020.02.011

The language of lambda terms $\lambda\text{-}term(Var, \Sigma_{\lambda_Z})$ is such that it has a signature $\Sigma_{\lambda_Z} := \bigcup_{x \in Var} \{\lambda x.\} \cup \{(.)\} \cup Z$ and is recursively defined as $AL(Var, \Sigma_Z)$, but adding that if $p \in \lambda\text{-}term(Var, \Sigma_{\lambda_Z})$ then $\lambda x.p \in \lambda\text{-}term(Var, \Sigma_{\lambda_Z})$. This definition implies that $AL(Var, Z) \subset \lambda\text{-}term(Var, \Sigma_{\lambda_Z})$.

The concept of bound and free variable are those that are defined in [2]. To simplify notation, given $t \in \lambda\text{-}term(Var, \Sigma_{\lambda_Z})$, we will write $x \in t$ and $x \in Var(t)$ to say that x occurs as a free variable and occurs as a free or bound variable respectively, in t.

On $\lambda\text{-}term(Var, \Sigma_{\lambda_Z})$, if $x \in Var$ and $p \in \lambda\text{-}term(Var, \Sigma_{\lambda_Z})$, the simple substitution operator is defined as follows

- $x < p|x > := p$ if $x \in Var$,
- $y < p|x > := y$ if $y \in Z$ or, $y \in Var$ and $x \neq y$,
- $(qr) < p|x > := q < p|x > r < p|x >$,
- $(\lambda x.q) < p|x > := \lambda x.q$,
- $(\lambda y.q) < p|x > := \lambda y.(q < p|x >)$ if $x \neq y$.

In this way since $AL(Var, \Sigma_Z) \subset \lambda\text{-}term(Var, \Sigma_{\lambda_Z})$, then $< p|x >$ is also defined in $AL(Var, Z)$.

In lambda calculus it is usual to consider that two equal terms, except for the name of their bound variables, are equivalent. This relationship is denoted $\equiv_\alpha$ and recursively defined as:

- If $u \in Var \cup Z$: $u \equiv_\alpha u'$ if and only if $u = u'$,
- If $u = pq$: $u \equiv_\alpha u'$ if and only if $u' = p'q'$, with $p \equiv_\alpha p'$ and $q \equiv_\alpha q'$
- If $u = \lambda x.v$: $u \equiv_\alpha u'$ if and only if $u' = \lambda y.v'$ with

$$(\exists A \subseteq Var \wedge \ A \text{ finite}) \text{ such that}$$

$$(\forall z \in Var \setminus A)(v < z|x > \equiv_\alpha v' < z|y >) \ .$$

We denote as Λ the set $\lambda\text{-}term(Var, \Sigma_{\lambda_Z})/\equiv_\alpha$, and given $p, q \in \Lambda$ and $x \in Var$, we define the α substitution $q[p|x]$, as the class of Λ, of the terms $q' < p'|x >$, where q' and p' are in the class of q and p respectively, and q' has names of its bound variables, which avoid variable capture. If in q there are no λ abstractions, then $[p|x]$ behaves isomorphic to $< p|x >$, so to unify notation, we will use $[p|x]$ to also refer to the operator $< p|x >$ in $AL(Var, \Sigma_Z)$. The concept of reduction $\rightarrow_\beta$ in Λ, is the same as in [3].

A combinator is a lambda term without free variables and without constants. Usually some combinators are abbreviated by constant symbols of Z, when this is the case, it is said that $AL(Var, \Sigma_Z)$ is a combinatory logic language and is denoted rather as $CL(Var, \Sigma_Z)$.

On $CL(Var, \Sigma_Z)$ the rewriting relation $\rightarrow_0$ is defined such that, if $\mathcal{C} \in Z$ is a combinator, then $\langle \mathcal{C}p_1p_2 \dots p_n, t_2 \rangle \in \rightarrow_0$ if and only if, n is the least natural, such that it is possible to apply several β reductions to $\mathcal{C}p_1p_2 \dots p_n$, only in redex caused

by the λ abstractions of $\mathcal{C}$, until all the sub-term $\mathcal{C}$ disappears, which results in the term t_2.

For example, the combinator $S := \lambda x.\lambda y.\lambda z.(xz)(yz)$ fulfills that 3 is the least natural to make $Sp_1p_2p_3 \to_\beta (\lambda y.\lambda z.(p_1z)(yz))p_2p_3 \to_\beta (\lambda z.(p_1z)(p_2z))p_3 \to_\beta (p_1p_3)(p_2p_3)$ where all the sub-term S disappeared. Using the above definition, it is true that $Sp_1p_2p_3 \to_0 (p_1p_3)(p_2p_3)$.

Other commonly used combinators are $I := \lambda x.x$, $K := \lambda x.\lambda y.x$, $B := \lambda x.\lambda y.\lambda z.x(yz)$ and $C := \lambda x.\lambda y.\lambda z.xzy$.

The relation $\to$ in $CL(Var, \Sigma_Z)$ is the smallest one that contains $\to_0$ and pass to the context (terminology of [3] in French). To pass to the context means that, if $t \to t'$, then $\lambda x.t \to \lambda x.t'$, $ut \to ut'$ and $tu \to t'u$.

The central definition of this work is the bracket abstraction of a single variable, which is a syntactic operator that given $x \in Var$ and combinatory term $t \in CL(Var, \Sigma_Z)$, returns a new term in $CL(Var, \Sigma_Z)$, denoted $[x]t$, which satisfies:

- $FreeVar([x]t) = FreeVar(t) \setminus \{x\}$
- $([x]t)p \to^* t[p|x]$. (The abbreviation **BA** will be used to refer to this property)

Remark 1.1 Some authors use $[x]t =_{\beta\eta} \lambda x.t$ instead of **BA** for the previous definition.

The original idea of the bracket abstraction was introduced in [4], which although in [4] there is no explicitly an algorithm to calculate $[x]t$, a calculation strategy can be implicitly inferred. Although it was Curry [5] who made this algorithm explicit, it has been called by the community as the Schönfinkel algorithm (abbreviated with SH). This algorithm is defined recursively as follows:

- $[x]x := I$,
- $[x]p = Kp$ if $x \notin p$,
- $[x]px = p$ if $x \notin p$ (η rule),
- $[x]pq = C([x]p)q$ if $x \in p$ and $x \notin q$,
- $[x]pq = Bp([x]q)$ if $x \notin p$ and $x \in q$,
- $[x]pq = S([x]p)([x]q)$ if $x \in p$ and $x \in q$.

(When there is ambiguity about which rule to use, the one that is listed first is used)

Over the years, other algorithms were defined in different combinator systems, each with certain properties, and in this work, to differentiate one algorithm from another, $[x]_A$ will be written where sub-index A indicates the algorithm used. For example, in [5], in order to simplify the foundations of combinatory logic, a simpler algorithm (which he called fab) was defined as follows:

- $[x]_{fab}x := I$,
- $[x]_{fab}p = Ky$ if $p \in Var$ and $p \neq x$,
- $[x]_{fab}pq = S([x]p)([x]q)$.

By replacing I in the first rule with SKK, we have the Hilbert algorithm (H), named for being isomorphic (by Curry-Howard correspondence) to the algorithm that converts the natural deduction inference rule, into a derivation in the Hilbert system. Also a variant to fab was made in the same book of Curry [5], called abf, is defined by changing the condition of the second rule by $x \notin p$. A variant to abf was made in the book of Hindley and Seldin (HS) [6] in which add the rule η to abf.

Functional languages are usually interpreted in a combinator machine, so algorithms for computing bracket abstractions are important for the implementation of functional languages, because they induce a $<>_A$: $\lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}}) \to CL(Var, \Sigma_Z)$ translation, defined as follows:

$$< p >_A = p \text{ if } p \in Var \text{ or } p \in Z'$$
$$< pq >_A = < p >_A < q >_A$$
$$< \lambda x.p >_A = [x]_A < p >_A$$
(Z is Z' plus the symbols representing combinators)

The properties studied in this work on bracket abstractions and different translations to combinators, are the following:

WD: If $p, q \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$ then $p \equiv_\alpha q \Rightarrow < p >_A = < q >_A$

SC: If $q, t \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$ and $x \in Var$ then $< p[t|x] >_A = < p >_A [< t >_A |x]$

SBA: $([x]_A p)t \to p[t|x]$

I: If $p, q \in \lambda - term(Var, \Sigma_{\lambda_{Z'}})$ then $< p >_A = < q >_A \Rightarrow p \equiv_\alpha q$.

The **WD** property is fulfilled by all bracket abstraction, and states that an Λ translation would be **W**ell **D**efined as follows:

$$<>_\Lambda : \Lambda \to CL(Var, \Sigma_Z)$$

$$< p >_\Lambda = < p' >$$
with p' any representative of p

The **SC** property states that the **S**ubstitution **C**ommutes with respect to $<>_A$. The **SBA** property is a **S**tronger property than **BA**. Property **I** states that the translation $<>_\Lambda$, with domain in Λ, is **I**njective.

1.1 Contribution

As mentioned in the previous section, there are several algorithms for bracket abstraction, but there is no unified theory to generalize all these algorithms, in fact the properties described in the previous section are demonstrated from scratch for each algorithm and combinator system. In this way, these types of demonstrations are repeated over and over throughout the bibliography.

In this work, a recursive algorithm scheme is defined that defines in general, an infinite family of bracket abstractions, where all the bracket abstraction of the bibliography referenced here are found. On this scheme a unified theory is developed

to demonstrate the properties described in the previous section, indicating for each of them, which are the hypotheses that must be assumed on the scheme, for the property to be true.

The defined scheme can be used as a basis when designing a new abstraction algorithm, since depending on the desired properties, the scheme and the hypotheses of the theorems that guarantee the compliance of the properties, would be a criterion to take in mind, during the design of the new algorithm.

On the other hand, here are criteria that the recursive scheme must have, in order for **I** to be fulfilled, a property that is apparently irrelevant for the compilation of functional languages, and therefore has not been studied by other authors. However, it is relevant together with the **SBA** property, since they *almost* establish an isomorphism of rewriting systems between Λ and the image of $<>_\Lambda$. With the bracket abstraction of Broda and Damas (which will be denoted $[x]_{BD}$), it is shown in [7], that **SBA** is almost fulfilled (see Remark 4.17), but this paper shows that eliminating the η rule of this bracket abstraction (algorithm that will be denoted $BD_{-\eta}$), then **SBA** is fulfilled. With the **I** and **WD** properties, in Broda and Damas's bracket abstraction without η rule, a lambda calculus normalization algorithm was designed, and implemented in the web application [8].

Property **I** and **WD** suggest an algorithm to decide whether two terms $p, q \in \lambda\text{-}term(Var)$ are α-equivalent. This algorithm consists in computing the image of both terms by the function $<>$, and if those are equal, then they are α-equivalent and otherwise they are not. This algorithm is the one used in [8], to avoid that two α-equivalent versions of the same term are stored in the terms database.

Additionally for those bracket abstraction that comply with **I**, a $<>^{-1}$ algorithm is enunciated to calculate the inverse of $< t >$. This algorithm was used in the web application [9], which is a proof assistant for the calculative logic, where each theorem t was coded as an λ-term, so that, thanks to **I**, $< t >$ uniquely indexes t, regardless of the name of the bound variables. In this way $< t >$ is stored in the database, but to the user, $<< t >>^{-1}$ is presented. This type of techniques thanks to **I**, are relevant, since the use of combinators for interactive and automatic theorem proving is a subject of study recently.

1.2 Related Works

The algorithms H, fab, abf, SH are not particularly efficient in terms of the size of the term $[x]_A t$ with respect to the size of t. For example, fab and H are exponential and abf is $O(n^3)$, so they are not useful for implementing functional languages. In [10,11], the size of $[x]_A t$ for different algorithms is analyzed.

The first improvement to Schönfinkel's algorithm was due to Turner [12], who defined optimization rules that were later used for the implementation of application languages [13]. The complexity in the worst case of the Turner algorithm is $O(n^2)$ [11] and on average $O(n^{3/2})$ [14]. There are several forms of the Turner algorithm, because its formulation contains ambiguities. In [15], different interpretations of the Turner algorithm is studied, and the one based on recursive equations with side conditions, is the algorithm taken for this job.

The advance in implementation techniques in functional languages meant an advance for the bracket abstraction theory. Bunder [16] discusses different extentions of the Turner algorithm. Then in [17] it is affirmed that using the combinator $B^* := \lambda fxyz.f(x(yz))$ instead of $B' := \lambda kxyz.kx(yz)$, in one of its optimization rules, the Turner algorithm is improved. However other ways of implementing application languages became standard, because produced faster implementations, among these techniques are those of super combinators [18].

A radical change in the way of abstraction of variables was due to the director Strings [19,20], although this was a formal system other than combinatorial logic, it was a precursor to the iconic representation. A significant contribution in the design of algorithms for the abstraction of variables in combinatorial logic was made by Stevens [21], who began using iconic representation to represent combinators, where the combinator's behavior could be read from his representation. Later Broda and Damas [7] defined a combinators base with iconic representation and an algorithm of abstraction of complexity $O(n^2)$ in the worst case. Finally, Antoni Diller [22], defined combinators with iconic representation and an algorithm of abstraction of complexity $O(a^2n)$ in the worst case, when a is the number of variables abstracted.

The properties **WD**, **BA**, **SBA**, **SC** and **I** were studied in the master thesis [23], for the algorithms H, abf, HS, SH and BD separately. In the present work the same study is done but in the framework of a unified theory of all abstraction algorithms. An application of the **WD**, **I** and **SBA** properties, in a modified Broda and Damas bracket abstraction, is found in the grade thesis [24], which explains the details of an algorithm to evaluate lambda calculus, using these properties and the details of a implementation located in [8].

Recently the use of combinators and bracket abstraction in interactive or automatic theorem proving has been studied. For example, Jacques Carette and Russell O'Connor in work [25], rewritten almost entirely in [26], has suggested making use of combinators to represent formulas, with the intention of generating a database of theorems scalable and without redundancy. His proposal is to use combinators, to represent the application structure of the mathematical statements and theories, in order to identify and keep common structures in the different mathematical theories.

From an experimental point of view, in [27], a comparison is made of the performance of several of the bracket abstractions algorithms studied here, but in the context of abstraction of variables in first-order mathematical theories, to accelerate the build of formal proofs, in an interactive theorem proving.

2 General recursion scheme

In the bracket abstraction algorithms presented in the introduction, it can be seen that $[x]p$ depends on some sub-terms of p. and the position of the sub-terms to use depends on how it is p. For this reason it is convenient to formalize the concepts of positions and replacement in a subterm position.

Definition 2.1 The positions $Pos(p)$ of a term p are defined as the set of strings

on the positive integer alphabet, recursively defined as:

- If $p \in Var$ or $p \in Z$ then $Pos(p) = \{\epsilon\}$, where ϵ denotes the empty string.
- If p is of the form qr, then

$$Pos(p) = \{\epsilon\} \cup \{1\sigma | \sigma \in Pos(q)\} \cup \{2\sigma | \sigma \in Pos(r)\}$$

The positions p and q are said to be parallel ($p||q$) iff they are incomparable with respect to the prefix order.

Definition 2.2 If $\sigma \in Pos(p)$, then the sub-term of p in the position σ is defined, denoted by $p|_\sigma$, recursively:

- $p|_\epsilon = p$
- If $p = qr$, then $p|_{1\sigma} = q|_\sigma$
- If $p = qr$, then $p|_{2\sigma} = r|_\sigma$

Definition 2.3 If $\sigma \in Pos(p)$, then $p[t]_\sigma$ is defined as the replacement by t of the sub-term at the position σ of p. Recursively:

- $p[t]_\epsilon = t$,
- If $p = qr$ then $p[t]_{1\sigma} = q[t]_\sigma r$
- If $p = qr$ then $p[t]_{2\sigma} = q(r[t]_\sigma)$

If $\sigma_1, \ldots, \sigma_k$ is a list of positions, then $p[t_1, \ldots, t_k]_{\sigma_1,\ldots,\sigma_k}$ is an abbreviation of $((p[t_1]_{\sigma_1})[t_2]_{\sigma_2}) \ldots [t_k]_{\sigma_k}$.

Definition 2.4 $|p|q$ it is an abbreviation of the term recursively defined as:

- $|p|(qt) := |pq|t$, $|p|q := pq$ y $|p|\epsilon := p$.

The previous definition is for short that $|p|w$ is associated to the left taking as arguments all the applications of w associated to the right. For example $|p|(w_1(w_2(w_3w_4)))$ is an abbreviation of $pw_1w_2w_3w_4$.

To state a general recursive algorithm scheme, to define a bracket abstraction, a list of positions σ_j will be used, to indicate which are the sub-terms that will be used of p, when p has a certain form or belongs to a certain set A_i.

Let $CL(Var, \Sigma_Z)$ be a combinatory logic, let m and k be such that $1 \leq k \leq m$ and let $\{A_i\}_{i=1}^m$ be a family of subsets of $Var \times CL(Var, \Sigma_Z)$ such that:

- $\cup_i A_i = Var \times CL(Var, \Sigma_Z)$
- For all x exists i with $1 \leq i \leq k$ such that $(x, p) \in A_i$ when p is variable or constant
- Let $y, z \in Var$. If i is the smallest index such that $(x, p) \in A_i$ and $x \notin \{y, z\}$ then i is the smallest index such that $(x, p[z|y]) \in A_i$
- Let $z \in Var$. If i is the smallest index such that $(x, p) \in A_i$ and $z \notin p$ then i is the smallest index such that $(z, p[z|x]) \in A_i$

For $1 \leq i \leq m$, let σ^i be lists of positions of length n_i. σ^i_j will be denoted to the

jth element of the ith list of positions σ^i. If $i > k$ then no position σ^i_j is ϵ.

For $1 \leq i \leq m$, let be $t_i : \Pi_{i=1}^{n'_i} CL(Var, \Sigma_Z) \to CL(Var, \Sigma_Z)$ where $n'_i = n_i$ when $1 \leq i \leq k$ and $n'_i = 2n_i$ when $k+1 \leq i \leq m$. The functions t_i are such that:

- are homomorphisms with respect to the substitution operator $[z|y]$ ($z \in Var$) i.e. $t_i(p_1, \ldots, p_{n'_i})[z|y] = t_i(p_1[z|y], \ldots, p_{n'_i}[z|y])$.
- if $(x,p) \in A_i$, then $x \notin t_i(p|_{\sigma^i_1}, \ldots, p|_{\sigma^i_{n_i}})$ when $1 \leq i \leq k$
- if $(x,p) \in A_i$, then $x \notin t_i(p|_{\sigma^i_1}, \ldots, p|_{\sigma^i_{n_i}}, r_1, \ldots, r_{n_i})$ when $k+1 \leq i \leq m$ and $x \notin r_j \in CL(Var, \Sigma_Z)$.

Definition 2.5 To define a bracket abstraction, the general recursive algorithm scheme is defined, as follows:

$$[x]p = t_1(p|_{\sigma^1_1}, \ldots, p|_{\sigma^1_{n_1}}) \text{ if } (x,p) \in A_1$$

$$\vdots\vdots$$

$$[x]p = t_k(p|_{\sigma^k_1}, \ldots, p|_{\sigma^k_{n_k}}) \text{ if } (x,p) \in A_k$$

$$[x]p = t_{k+1}(p|_{\sigma^{k+1}_1}, .., p|_{\sigma^{k+1}_{n_{k+1}}}, [x](p|_{\sigma^{k+1}_1}), .., [x](p|_{\sigma^{k+1}_{n_{k+1}}})) \text{ if } (x,p) \in A_{k+1}$$

$$\vdots\vdots$$

$$[x]p = t_m(p|_{\sigma^m_1}, \ldots, p|_{\sigma^m_{n_m}}, [x](p|_{\sigma^m_1}), \ldots, [x](p|_{\sigma^m_{n_m}})) \text{ if } (x,p) \in A_m$$

Remark 2.6 In case the pair (x,p) belongs to several A_i's, then the clause that is listed first will have priority. Also the possibility that $m = \infty$, is considered (like in Definition 3.11).

3 Classic Bracket Abstractions and General Recursive Scheme

This section shows how the bracket abstractions of the bibliography adapts to the general recursive scheme.

Definition 3.1 [fab [5]] The fab algorithm can be defined by instantiating the general scheme with $k = 2$, $t_1(p) = I$, $t_2(p) = Kp$, $t_3(p,q,r,w) = Srw$, with the lists of positions $\sigma^1 := \epsilon$, $\sigma^2 := \epsilon$ and $\sigma^3 := 1, 2$ and with the sets
$A_1 := \{(x,p) \in A | x = p\}$,
$A_2 := \{(x,p) \in A | (p \in Var \wedge p \neq x) \vee p \in Z\}$ and
$A_3 := \{(x,p) \in A | (\exists q, r)(p = qr)\}$, where
$A := Var \times CL(Var, \Sigma_{Z' \cup \{I,S,K\}})$

Definition 3.2 [Hilbert (H)] Hilbert's algorithm (H) is defined as fab except that $t_1(p) = SKK$ and $A := Var \times CL(Var, \Sigma_{Z' \cup \{S,K\}})$.

Definition 3.3 [abf [3]] The abf algorithm is defined the same as fab except that $A_2 := \{(x,p) \in A | x \notin p\}$.

Definition 3.4 [Hindley, Lercher y Seldin (HS) [6]] The HS algorithm can be defined by instantiating the general scheme with $k := 3$, $t_1(p) = I$, $t_2(p) = Kp$,

$t_3(p,q) = p$, $t_4(p,q,r,w) = Srw$, with the same lists of fab plus $\sigma^4 := 1,2$ and with the sets A_1, A_2 as in abf,
$A_3 := \{(x,p) \in A | (\exists q)(p = qx \wedge x \notin q)\}$ and
$A_4 := \{(x,p) \in A | (\exists q,r)(p = qr)\}$, with A as in fab.

Definition 3.5 [Schönfinkel (SH) [4]] The SH algorithm is defined as HS except that $t_4(p,q,r,w) = Crq$, $t_5(p,q,r,w) = Bpw$, $t_6(p,q,r,w) = Srw$, $\sigma^5 := 1,2$, $\sigma^6 := 1,2$ and the sets
$A_4 := \{(x,p) \in A | (\exists q,r)(p = qr \wedge x \in q \wedge x \notin r)\}$,
$A_5 := \{(x,p) \in A | (\exists q,r)(p = qr \wedge x \notin q \wedge x \in r)\}$
$A_6 := \{(x,p) \in A | (\exists q,r)(p = qr \wedge x \in q \wedge x \in r)\}$ where
$A := Var \times CL(Var, \Sigma_{Z' \cup \{I,S,K,B,C\}})$.

Definition 3.6 [Schönfinkel modified ($SH_{-\eta}$)] The algorithm $SH_{-\eta}$ is defined the same as SH but eliminating the η rule. This corresponds to making $k := 2$, removing the t_3, σ^3, A_3 from SH and subtracting 1 from the indices of the functions t_i, sets A_i and lists σ^i with $i > 3$.

Definition 3.7 [Turner (T) [12]] The T algorithm is defined as SH except that $k := 4$, $t_4(p,q) = Cpq$, $t_5(p,q,r,w) = Spw$, $t_6(p,q,t,r,w,s) = B'pqs$, $t_7(p,q,t,r,w,s) = C'pwt$, $t_8(p,q,t,r,w,s) = S'pws$, $\sigma^4 := 11,2$, $\sigma^5 := 11,2$, $\sigma^6 := 11,12,2$, $\sigma^7 := 11,12,2$, $\sigma^8 := 11,12,2$, and t_9, t_{10}, t_{11}, σ^9, σ^{10}, σ^{11}, A^9, A^{10}, A^{11} as t_4, t_5, t_6, σ^4, σ^5, σ^6, A^4, A^5, A^6 of SH respectively, and with the sets
$A_4 := \{(x,p) \in A | (\exists q,r)(p = qxr \wedge x \notin qr)\}$
$A_5 := \{(x,p) \in A | (\exists q,r)(p = qxr \wedge x \notin q \wedge x \in r)\}$
$A_6 := \{(x,p) \in A | (\exists q,s,r)(p = qsr \wedge x \notin qs \wedge x \in r)\}$
$A_7 := \{(x,p) \in A | (\exists q,s,r)(p = qsr \wedge x \notin qr \wedge x \in s)\}$
$A_8 := \{(x,p) \in A | (\exists q,s,r)(p = qsr \wedge x \notin q \wedge x \in s \wedge x \in r)\}$ where
$A := Var \times CL(Var, \Sigma_{Z' \cup \{I,S,K,B,C,S',B',C'\}})$.

Definition 3.8 [Broda and Damas (BD) [7]] The Broda and Damas indexes $\mathscr{A}$ are strings of letters b and c with the following formation rules: $\epsilon \in \mathscr{A}$ (ϵ is the empty string) and if $\alpha_1, \alpha_2 \in \mathscr{A}$, then $c\alpha_1, b\alpha_1, (\alpha_1,\alpha_2) \in \mathscr{A}$. The combinators of Broda and Damas are constants of the form Φ_K or Φ_α with $\alpha \in \mathscr{A}$. The set of these combinators is denoted $\mathscr{K}$.

The BD algorithm is defined the same as SH but taking $A := Var \times CL(Var, \Sigma_{Z' \cup \mathscr{K}})$ with the functions $t_1(p) = \Phi_\epsilon$, $t_2(p) = \Phi_K p$, $t_3(p,q) = p$,

$$t_4(p,q,r_1,r_2) = \begin{cases} |\Phi_{c\alpha}|(qw) \ if \ r_1 = |\Phi_\alpha|w \\ \Phi_\epsilon \qquad\qquad otherwise \end{cases}$$

$$t_5(p,q,r_1,r_2) = \begin{cases} |\Phi_{b\alpha}|(pw) \ if \ r_2 = |\Phi_\alpha|w \\ \Phi_\epsilon \qquad\qquad otherwise \end{cases}$$

$$t_6(p,q,r_1,r_2) = \begin{cases} ||\Phi_{(\alpha_1,\alpha_2)}|w_1|w_2 \qquad if \qquad r_1 = |\Phi_{\alpha_1}|w_1 \quad and \quad r_2 = |\Phi_{\alpha_2}|w_2 \\ \Phi_\epsilon \qquad\qquad\qquad otherwise \end{cases}$$

This is the expedited definition of Bunder [28].

Remark 3.9 In the conditions of the pieces of the previous functions it is contemplated that w, w_1, w_2, α, α_1 and α_2 can be ϵ

Definition 3.10 [Broda and Damas modified ($BD_{-\eta}$)] The $BD_{-\eta}$ algorithm is defined the same as BD but eliminating the η rule. This corresponds to making $k := 2$, removing the t_3, σ^3, A_3 from BD and subtracting 1 the indices of the functions t_i, sets A_i and lists σ^i with $i > 3$.

Definition 3.11 [Diller (D) [22]] The combinators of Diller, called *yn-strings*, are strings of letters y and n. Algorithm D is defined by instantiating the general scheme with $k := 2$, $\sigma^1, \sigma^2 := \epsilon, \epsilon$, $t_1(p) = yI$, $t_2(p) = np$, and A_1, A_2 as fab. Defining $J_j := \frac{3^j-3}{2}$ and $q_1q_2 \dots q_j$ as $(i - J_j)$ written in base 3 when $J_j \leq i < J_{j+1}$, it is instantiated $m := \infty$ $n_i := j$, $\sigma^i := \overbrace{1 \dots 1}^{j-1 \ times}, \overbrace{1 \dots 1}^{j-2 \ times} 2, \dots, 12, 2$ for all $i > 2$ and $t_i(p_1, \dots, p_j, r_1, \dots, r_j) = \phi_{q_1} \dots \phi_{q_j} Q^1_{q_1} \dots Q^j_{q_j}$ where $\phi_0 = y$, $Q^l_0 = I$, $F^l_0 := (p_l = x)$, $\phi_1 = y$, $Q^l_1 = r_l$, $F^l_1 := (x \in p_l \wedge p_l \neq x)$, $\phi_2 = n$, $Q^l_2 = p_l$, $F^l_2 := (x \notin p_l)$. And the sets $A_i := \{(x, p) \in A |$
$(\exists p_1, \dots, p_j)(p = p_1 \dots p_j \wedge p_1 \in Var \cup Z \wedge F^1_{q_1} \wedge .. \wedge F^j_{q_j})\}$ with $A := Var \times CL(Var \times \Sigma_{Z' \cup \{I\} \cup yn\text{-}strings})$.

Each of the functions t_i are homomorphisms with respect to $[z|x]$. For example in SH, is fulfilled $t_5(p, q, r, w)[z|y] = (Bpw)[z|y] = Bp[z|y]w[z|y] = t_5(p[z|y], q[z|y], r[z|y], w[z|y])$. For the rest of the t_i, similar proves can be made.

4 Properties of Bracket Abstractions

4.1 $<>_\Lambda$ *is Well Defined (***WD***)*

This section will show that with the bracket abstraction of the general recursion scheme $[x]$, the **WD** property is satisfied

Lemma 4.1 *If $x, y \in Var$, $p \in CL(Var, \Sigma_Z)$ then $x \notin [x]p$, and if $x \neq y \notin p$ then $y \notin [x]p$*

Proof. Note that if $y \notin p$ then $y \notin p|_\sigma$ for any position σ. (*)
The proof is by structural induction on p.

If $p \in Var \cup Z$:

By definition the lowest index such that $(x, p) \in A_i$, fulfills $i \leq k$, then $[x]p = t_i(p|_{\sigma^i_1}, \dots, p|_{\sigma^i_{n_i}}) \overset{def}{\not\equiv} x$. Furthermore, if $x \neq y \notin p$, then assuming $y \in [x]p$ leads to a contradiction because taken $z \neq y$, then $t_i(p|_{\sigma^i_1}[z|y], \dots, p|_{\sigma^i_{n_i}}[z|y]) \overset{(*)}{=} t_i(p|_{\sigma^i_1}, \dots, p|_{\sigma^i_{n_i}}) = [x]p \overset{z \neq y \in [x]p}{\neq} ([x]p)[z|y] = t_i(p|_{\sigma^i_1}, \dots, p|_{\sigma^i_{n_i}})[z|y]$ but impossible because t_i is an homomorphism with respect $[z|y]$. Therefore $y \notin [x]p$.

If $p \notin Var \cup Z$ then it separates in cases:

Case 1: The smallest index i such that $(x,p) \in A_i$, fulfills $1 \leq i \leq k$. The proof is the same as the base case

Case 2: The smallest index i such that $(x,p) \in A_i$, fulfills $k+1 \leq i \leq m$
$[x]p = t_i(p|_{\sigma_1^i}, \ldots, p|_{\sigma_{n_i}^i}, [x]p|_{\sigma_1^i}, \ldots, [x]p|_{\sigma_{n_i}^i})$ since by **I.H.** (**I**nductive **H**ypothesis) $x \notin [x]p|_{\sigma_j^i}$, then $x \notin t_i(p|_{\sigma_1^i}, \ldots, p|_{\sigma_{n_i}^i}, [x]p|_{\sigma_1^i}, \ldots, [x]p|_{\sigma_{n_i}^i})$ by definition of t_i. Furthermore, if $x \neq y \notin p$, then assuming $y \in [x]p$ leads to a contradiction because with $z \neq y$, $t_i(p|_{\sigma_1^i}[z|y]..p|_{\sigma_{n_i}^i}[z|y], ([x]p|_{\sigma_1^i})[z|y]..([x]p|_{\sigma_{n_i}^i})[z|y]) \overset{(*),I.H.}{=} t_i(p|_{\sigma_1^i}, \ldots, p|_{\sigma_{n_i}^i}, [x]p|_{\sigma_1^i}, \ldots, [x]p|_{\sigma_{n_i}^i}) = [x]p \overset{z \neq y \in [x]p}{\neq} ([x]p)[z|y] = t_i(p|_{\sigma_1^i}, \ldots, p|_{\sigma_{n_i}^i}, [x]p|_{\sigma_1^i}, \ldots [x]p|_{\sigma_{n_i}^i})[z|y]$ but impossible because t_i is an homomorphism with respect $[z|y]$. Therefore $y \notin [x]p$.

□

Lemma 4.2 *If $x, z \in Var$, $p \in CL(Var, \Sigma_Z)$ and $z \notin p$ then $[z](p[z|x]) = [x]p$*

Proof. Since $z \notin p$ then $z \notin p|_\sigma$ for any position σ. (*)
It is separated by cases:
Case 1: $z = x$

$[z](p[z|x]) \overset{x=z}{=} [x](p[x|x]) = [x]p$

Case 2: $z \neq x$. The proof is by structural induction on p

If $p \in Var \cup Z$ then, by definition the smallest index such that $(x,p) \in A_i$, fulfills $1 \leq i \leq k$, then

$[x]p \overset{Lemma\ 4.1}{=} ([x]p)[z|x] \overset{def}{=} t_i(p|_{\sigma_1^i}, \ldots, p|_{\sigma_{n_i}^i})[z|x] \overset{t_i\ is\ homo}{=} t_i(p|_{\sigma_1^i}[z|x], \ldots, p|_{\sigma_{n_i}^i}[z|x])$

On the other hand, i is the smallest index such that $(z, p[z|x]) \in A_i$ (by definition of A_i), therefore the computation of $[z](p[z|x])$ takes the rule t_i. For this reason

$[z](p[z|x]) = t_i(p[z|x]|_{\sigma_1^i}, \ldots, p[z|x]|_{\sigma_{n_i}^i}) = t_i(p|_{\sigma_1^i}[z|x], \ldots, p|_{\sigma_{n_i}^i}[z|x])$

$\therefore [x]p = [z](p[z|x])$

If $p \notin Var \cup Z$ then it separates in cases

Case 2.2: The smallest index i such that $(x,p) \in A_i$, fulfills $1 \leq i \leq k$. The proof is like the base case

Case 2.1: The smallest index i such that $(x,p) \in A_i$, fulfills $k+1 \leq i \leq m$

$[x]p \overset{Lemma\ 4.1}{=} ([x]p)[z|x] \overset{def}{=} t_i(p|_{\sigma_1^i}, \ldots, p|_{\sigma_{n_i}^i}, [x]p|_{\sigma_1^i}, \ldots, [x]p|_{\sigma_{n_i}^i})[z|x] \overset{t_i\ is\ homo}{=} t_i(p|_{\sigma_1^i}[z|x]..p|_{\sigma_{n_i}^i}[z|x], ([x]p|_{\sigma_1^i})[z|x]..([x]p|_{\sigma_{n_i}^i})[z|x]) \overset{Lemma 4.1}{=}$

$t_i(p|_{\sigma_1^i}[z|x], \ldots, p|_{\sigma_{n_i}^i}[z|x], [x]p|_{\sigma_1^i}, .., [x]p|_{\sigma_{n_i}^i})$

On the other hand, i is the smallest index such that $(z, p[z|x]) \in A_i$ (by definition of A_i), then the computation of $[z](p[z|x])$ takes the rule t_i. Therefore $[z](p[z|x]) = t_i(p[z|x]|_{\sigma_1^i}, \ldots, p[z|x]|_{\sigma_{n_i}^i}, [z](p[z|x]|_{\sigma_1^i}), \ldots, [z](p[z|x]|_{\sigma_{n_i}^i}))$
$= t_i(p|_{\sigma_1^i}[z|x].., p|_{\sigma_{n_i}^i}[z|x], [z](p|_{\sigma_1^i}[z|x]).., [z](p|_{\sigma_{n_i}^i}[z|x])) \overset{(*) \text{ and } I.H}{=}$
$t_i(p|_{\sigma_1^i}[z|x] \ldots, p|_{\sigma_{n_i}^i}[z|x], [x]p|_{\sigma_1^i}.., [x]p|_{\sigma_{n_i}^i})$

$\therefore [x]p = [z](p[z|x])$

□

Lemma 4.3 *If $x, y, z \in Var$, $p \in CL(Var, \Sigma_Z)$ and $y \notin \{x, z\}$ then $[y](p[z|x]) = ([y]p)[z|x]$*

Proof. By structural induction on p:
If $p \in Var \cup Z$:

By definition the smallest index i such that $(y, p) \in A_i$, fulfills $1 \leq i \leq k$, then $([y]p)[z|x] \overset{def}{=} t_i(p|_{\sigma_1^i}, \ldots, p|_{\sigma_{n_i}^i})[z|x] \overset{t_i \text{ is homo}}{=} t_i(p|_{\sigma_1^i}[z|x], \ldots, p|_{\sigma_{n_i}^i}[z|x]) = t_i(p[z|x]|_{\sigma_1^i}, \ldots, p[z|x]|_{\sigma_{n_i}^i})$.

On the other hand since i is the smallest index such that $(y, p[z|x]) \in A_i$ then $[y](p[z|x]) = t_i(p[z|x]|_{\sigma_1^i}, \ldots, p[z|x]|_{\sigma_{n_i}^i})$

$\therefore ([y]p)[z|x] = [y](p[z|x])$

If $p \notin Var \cup Z$ then it is separated in cases:

Case 1: The smallest index i such that $(y, p) \in A_i$, fulfills $1 \leq i \leq k$. The proof is the same as the base case

Case 2: The smallest index i such that $(y, p) \in A_i$, fulfills $k + 1 \leq i \leq m$

$([y]p)[z|x] \overset{def}{=} t_i(p|_{\sigma_1^i}, .., p|_{\sigma_{n_i}^i}, [y]p|_{\sigma_1^i}, .., [y]p|_{\sigma_{n_i}^i})[z|x]$
$\overset{t_i \text{ is homo}}{=} t_i(p|_{\sigma_1^i}[z|x].., p|_{\sigma_{n_i}^i}[z|x], ([y]p|_{\sigma_1^i})[z|x].., ([y]p|_{\sigma_{n_i}^i})[z|x])$
$\overset{I.H.}{=} t_i(p|_{\sigma_1^i}[z|x].., p|_{\sigma_{n_i}^i}[z|x], [y](p|_{\sigma_1^i}[z|x]).., [y](p|_{\sigma_{n_i}^i}[z|x]))$
$= t_i(p[z|x]|_{\sigma_1^i}.., p[z|x]|_{\sigma_{n_i}^i}, [y](p[z|x]|_{\sigma_1^i}).., [y](p[z|x]|_{\sigma_{n_i}^i}))$

On the other hand since i is the smallest index such that $(y, p[z|x]) \in A_i$ then $[y](p[z|x]) = t_i(p[z|x]|_{\sigma_1^i}.., p[z|x]|_{\sigma_{n_i}^i}, [y](p[z|x]|_{\sigma_1^i}).., [y](p[z|x]|_{\sigma_{n_i}^i}))$

$\therefore ([y]p)[z|x] = [y](p[z|x])$

□

Remark 4.4 The previous lemma is not generally true if z is not a variable, for example for Hilbert's bracket abstraction, we have:

$[y]_H(x[Kx|x]) = [y]_H(Kx) = S([y]_H K)([y]_H x) = S(KK)(Kx)$ On the other hand

$([y]_H x)[Kx|x] = (Kx)[Kx|x] = K(Kx)$

Thus

$[y]_H(x[Kx|x]) \neq ([y]_H x)[Kx|x]$

However, it is later determined that by adding conditions to the general recursion scheme, the substitution commutes with the bracket abstraction, regardless of whether the term to be substituted is not a variable.

With the three previous lemmas it is possible to prove:

Lemma 4.5 *If $r \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$ and $x, z \in Var$ such that $z \notin Var(r)$ then* $< r < z|x >>=< r > [z|x]$

Proof. By structural induction on r:
If $r \in Var$ and $r = x$:

$< r < z|x >>=< z >= z = x[z|x] = r[z|x] =< r > [z|x]$.

If $r \in Z'$ or, $r \in Var$ and $r \neq x$:

$< r < z|x >>=< r >= r = r[z|x] =< r > [z|x]$.

If $r = pq$ with $p, q \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$:

$< r < z|x >>=< pq < z|x >>=< p < z|x > q < z|x >>=< p < z|x >>< q < z|x >> \overset{z \notin Var(p) \wedge z \notin Var(q),\ I.H.}{=}$
$< p > [z|x] < q > [z|x] = (< p >< q >)[z|x] =< pq > [z|x] =< r > [z|x]$.

If $r = \lambda y.p$ with $y \in Var$ and $p \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$: It separates in two cases.

Case 1 $y = x$:

$< r < z|x >>=< (\lambda y.p) < z|x >>=< \lambda y.p >= [y] < p > \overset{Lemma\ 4.1}{=} ([y] < p >)[z|y] =$
$([y] < p >)[z|x] =< \lambda y.p > [z|x] =< r > [z|x]$.

Case 2 $y \neq x$:

$< r < z|x >>=< (\lambda y.p) < z|x >>= < \lambda y.(p < z|x >) >= [y] < p < z|x >> \overset{z \notin Var(p),\ I.H.}{=}$
$[y](< p > [z|x]) \overset{(z \notin Var(r) \wedge x \neq y \Rightarrow z \neq y \wedge x \neq y) \wedge Lemma\ 4.3}{=} ([y] < p >)[z|x] =< \lambda y.p > [z|x] =< r > [z|x]$.

□

Theorem 4.6 (WD) *If $p, q \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$ then*

$$p \equiv_\alpha q \Rightarrow < p >=< q >$$

Proof. By structural induction on p.
If $p \in Var \cup Z'$:

$p \equiv_\alpha q \overset{def}{\iff} p = q \overset{def}{\Rightarrow} < p >= p = q =< q >$.

If $p = rt$ with $r, t \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$:

$p \equiv_\alpha q \overset{def}{\iff} q = r't'$ with

$r', t' \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$, $r \equiv_\alpha r'$ and $t \equiv_\alpha t'$.

By inductive hypothesis
$< r >=< r' >$ and $< t >=< t' >$
$\Rightarrow$
$< r >< t >=< r' >< t' >$
$\Rightarrow$
$< p >=< rt >=< r >< t >=< r' >< t' >=< r't' >=< q >$.

If $p = \lambda x.r$ with $x \in Var$ and $r \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$:

$p \equiv_\alpha q \overset{def}{\iff} q = \lambda y.r'$ with $r' \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$, $y \in Var$ and there is finite $A \subseteq Var$ such that
$r < z|x >\equiv_\alpha r' < z|y >$ for all $z \in Var \setminus A$.

By Inductive Hypothesis $< r < z|x >>=< r' < z|y >>$ for all $z \in Var \setminus A$ and defining
$A' := A \cup \{x \in Var | x \in Var(< r >< r' > rr')\}$ then
$< r > [z|x] \overset{Lemma\ 4.5}{=} < r < z|x >>=$
$< r' < z|y >> \overset{Lemma\ 4.5}{=} < r' > [z|y]$ for all $z \in Var \setminus A'$
$\overset{[z]\ is\ a\ function}{\Rightarrow}$
$[z](< r > [z|x]) = [z](< r' > [z|y])$ for all $z \in Var \setminus A'$
$\overset{z \notin < r > \wedge z \notin < r' > \wedge\ Lemma\ 4.2}{\Rightarrow}$
$[x] < r >= [y] < r' >$
$\iff$
$< p >=< \lambda x.r >=< \lambda y.r' >=< q >$

□

Lemma 4.7 *If the variable x does not occur free in $t \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$, then $x \notin < t >$*

Proof. By structural induction on t and Lemma 4.1. □

4.2 **SC** *Property*

The **SC** property is not true in general, for example translating with the Hilbert bracket abstraction we have to

$$< (\lambda y.x)[\lambda y.x|x] >_H =< \lambda y.(x[\lambda y.x|x]) >_H =< \lambda y.\lambda y.x >_H =$$
$$[y]_H([y]_H x) = [y]_H(x[[y]_H x|x]) \overset{Remark\ 4.4}{\neq} ([y]_H x)[[y]_H x|x] =< \lambda y.x >_H [< \lambda y.x >_H |x]$$

However, we can add conditions to the general recursion scheme so that **SC** is true. These conditions are:

- The functions t_i are homomorphisms with respect to the operator $[t|y]$ where $t \in CL(Var, \Sigma_Z)$
- Let $y \in Var$ and $t \in CL(Var, \Sigma_Z)$. If i is the smallest index such that $(x, p) \in A_i$, $x \neq y$ and $x \notin t$ then i is the smallest index such that $(x, p[t|y]) \in A_i$.

Notation When an operator $[x]$ satisfies the conditions of the recursion scheme of the previous section, together with these conditions, $[x]$ is said to **fit to the strong recursion scheme**

Under these conditions Lemma 4.3 can be generalized

Lemma 4.8 *If $[x]$ fits the strong recursion scheme then. If $p \in CL(Var, \Sigma_Z)$, $y \notin t$ and $y \neq x$ then $[y](p[t|x]) = ([y]p)[t|x]$*

Proof. The proof is identical to that of Lemma 4.3 replacing z with t. □

With this it is easy to prove **SC**.

Theorem 4.9 (SC) *If $q, t \in \Lambda$ and $<>_\Lambda$ was defined using a bracket abstraction that fits with the strong recursion scheme, then $< q[t|x] >_\Lambda =< q >_\Lambda [< t >_\Lambda |x]$.*

Proof. By structural induction on q.
If $q \in Var$.

Case 1 $q = x$.

$< q[t|x] >_\Lambda =< x[t|x] >_\Lambda =< t >_\Lambda = x[< t >_\Lambda |x] =< x >_\Lambda [< t >_\Lambda |x] =< q >_\Lambda [< t >_\Lambda |x]$.

Case 2 $q \neq x$.

$< q[t|x] >_\Lambda =< q >_\Lambda = q = q[< t >_\Lambda |x] =< q >_\Lambda [< t >_\Lambda |x]$.

If $q = pr$ with $p, r \in \Lambda$.

$< q[t|x] >_\Lambda =< (pr)[t|x] >_\Lambda =< p[t|x]r[t|x] >_\Lambda =$
$< p[t|x] >_\Lambda < r[t|x] >_\Lambda \overset{I.H.}{=}$
$< p >_\Lambda [< t >_\Lambda |x] < r >_\Lambda [< t >_\Lambda |x] =$
$(< p >_\Lambda < r >_\Lambda)[< t >_\Lambda |x] =$
$< pr >_\Lambda [< t >_\Lambda |x] =< q >_\Lambda [< t >_\Lambda |x]$

If $q = \lambda y.p$ with $y \in Var$ and $p \in \Lambda$.

$< q[t|x] >_\Lambda =< (\lambda y.p)[t|x] >_\Lambda =< (\lambda y'.p') < t'|x >>$ where $y' \in Var$, with $y' \neq x$ and $p', t' \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$ such that $\lambda y'.p' \in q$, $t' \in t$ and $BoundVar(\lambda y'.p') \cap FreeVar(t') \neq \emptyset$.

$\Rightarrow$

$< q[t|x] >_\Lambda =< (\lambda y'.p') < t'|x >> \overset{y' \neq x}{=} < \lambda y'.(p' < t'|x >) >= [y'] < p' < t'|x >>=$
$[y'] < p[t|x] >_\Lambda \overset{I.H.}{=} [y'](< p >_\Lambda [< t >_\Lambda |x]) \overset{Lemma\ 4.8\ and\ 4.7}{=} ([y'] < p >_\Lambda)[< t >_\Lambda |x] \overset{Theorem\ 4.6}{=}$
$([y'] < p' >)[< t >_\Lambda |x] =< \lambda y'.p' > [< t >_\Lambda |x] \overset{Theorem\ 4.6}{=} < \lambda y.p >_\Lambda [< t >_\Lambda |x] =< q >_\Lambda [< t >_\Lambda |x]$

□

4.3 Reduction Property **BA**

All bracket abstraction must fulfills the property **BA**. However, in general, there is no deterministic algorithm that specify the reduction strategy to reduct $([x]p)t$ to $p[t|x]$. For example, in the case of $[x]_A$ with $A \in \{HS, SH\}$, we have to $([x]_A(Ix))t = It \to^0 It = (Ix)[t|x]$, but for another side, $([x]_A x)t$ is also equal to It and $It \to t = x[t|x]$. This prove that the term $([x]_A p)t$, by itself, in general is not enough to know the reduction strategy that leads to $p[t|x]$, although it is known that it exists said strategy.

This section is dedicated to explaining what additional conditions, the general recursion scheme needs, so that **BA** is true and the term $([x]p)t$, by itself, is enough to determine the reduction strategy that leads it to $p[t|x]$.

Theorem 4.10 (BA) *If a bracket abstraction fit the general recursion scheme, and additionally satisfies the following conditions:*

(i) *If $(x,p) \in A_i$ with $1 \le i \le k$, then there is a succession of reductions, independent of who (x,p) is in A_i, that converts $t_i(p|_{\sigma^i_1}, \dots, p|_{\sigma^i_{n_i}})r$ to $p[r|x]$,*

(ii) *If $(x,p) \in A_i$ with $k < i \le m$, then there is a succession of reductions, independent of who (x,p) is in A_i, that converts $t_i(p|_{\sigma^i_1}, \dots, p|_{\sigma^i_{n_i}}, [x]p|_{\sigma^i_1}, \dots, [x]p|_{\sigma^i_{n_i}})r$ to $p[([x]p|_{\sigma^i_{j_1}})r, \dots, ([x]p|_{\sigma^i_{j_l}})r]_{\sigma^i_{j_1},\dots,\sigma^i_{j_l}}$, where $Par := \{\sigma^i_{j_1}, \dots, \sigma^i_{j_l}\}$ is a subset of parallel positions of $\{\sigma^i_k\}_{k=1}^{n_i}$, such that "all occurrences of x in p are found in the subterms $p|_{\sigma^i_{j_1}}, \dots, p|_{\sigma^i_{j_l}}$" $(**)$*

then $([x]p)r \to^* p[r|x]$, and if additionally satisfies the following:

(iii) *If $[x]p = [x']p'$ and i is the smallest index such that $(x,p) \in A_i$ then i is the smallest index such that $(x',p') \in A_i$,*

then, the result of the succession of reductions over $([x]p)r$, defines a function $\kappa : Dom \to CL(Var, \Sigma_Z)$ where $Dom := \{qr | (\exists x \in Var)(q \in Img([x])) \wedge r \in CL(Var, \Sigma_Z)\}$, which fulfills $\kappa(([x]p)r) = p[r|x]$, and is computed with the following recursive rules:

- *$\kappa(([x]p)r) = q$ with $([x]p)r \overset{ith\ succ\ of\ reductions}{\to^*} q$ if $(x,p) \in A_i$ with $1 \le i \le k$ and,*
- *$\kappa(([x]p)r) = p[\kappa(([x]p|_{\sigma^i_{j_1}})r), .., \kappa(([x]p|_{\sigma^i_{j_l}})r)]_{\sigma^i_{j_1},\dots,\sigma^i_{j_l}}$ if $(x,p) \in A_i$ only with $k < i \le m$.*

Proof. Apply the same sequence of reductions to equal terms, results in equal terms $(***)$.

With the set of pairs $\kappa := \{t | (\exists x,p,r)((x,p) \in A \wedge r \in CL(Var, \Sigma_Z) \wedge \langle([x]p)r, p[r|x]\rangle = t)\}$, it proves by structural induction that $([x]p)r \to^* p[r|x]$ and κ is a function if (iii) (The latter will be done by proving that $p[r|x] = p'[r'|x']$ if $([x]p)r = ([x']p')r'$).

If $p \in Var \cup Z$ or $(x,p) \in A_i$ with $1 \le i \le k$

In both cases exists $i \le k$ such that $([x]p) = t_i(p|_{\sigma^i_1}, \dots, p|_{\sigma^i_{n_i}})$.

According to (i) $([x]p)r = t_i(p|_{\sigma_1}, \ldots, p|_{\sigma_{n_i}})r \to^* p[r|x]$. Secondly, if the third condition is fulfilled, $([x]p)r = ([x']p')r' \overset{\text{(iii)}}{\Rightarrow} t_i(p|_{\sigma_1^i}, \ldots, p|_{\sigma_{n_i}^i})r = t_i(p'|_{\sigma_1^i}, \ldots, p'|_{\sigma_{n_i}^i})r'$, applying the ith sequence of reductions on both sides $\overset{\text{item (i) and (***)}}{\Rightarrow} t_i(p|_{\sigma_1^i}, \ldots, p|_{\sigma_{n_i}^i})r \to^* p[r|x] = p'[r'|x'] \,{}^*\!\leftarrow t_i(p'|_{\sigma_1^i}, \ldots, p'|_{\sigma_{n_i}^i})r'$

If p is an application and the smallest index i such that $(x,p) \in A_i$, fulfills $k < i \leq m$

In this case

$[x]p = t_i(p|_{\sigma_1^i}, \ldots, p|_{\sigma_{n_i}^i}, [x]p|_{\sigma_1^i}, \ldots, [x]p|_{\sigma_{n_i}^i})$

According to (ii)

$([x]p)r \to^* p[([x]p|_{\sigma_{j_1}^i})r, \ldots, ([x]p|_{\sigma_{j_l}^i})r]_{\sigma_{j_1}^i,\ldots,\sigma_{j_l}^i}$

$\overset{\text{I.H. and } \sigma_{j_1},\ldots,\sigma_{j_l} \text{ are parallel positions}}{\to^*} p[p|_{\sigma_{j_1}^i}[r|x], \ldots, p|_{\sigma_{j_l}^i}[r|x]]_{\sigma_{j_1}^i,\ldots,\sigma_{j_l}^i}$

$\overset{p|_{\sigma_{j_1}^i},\ldots,p|_{\sigma_{j_l}^i} \text{ contains all ocurrences of } x \text{ in } p \text{ (**)}}{=} p[r|x]$. Secondly, if the third condition is fulfilled, $([x]p)r = ([x']p')r' \overset{\text{(iii)}}{\Rightarrow} t_i(p|_{\sigma_1^i}, .., p|_{\sigma_{n_i}^i}, [x']p|_{\sigma_1^i}, .., [x']p|_{\sigma_{n_i}^i})r = ([x]p)r = ([x']p')r' = t_i(p'|_{\sigma_1^i}, .., p'|_{\sigma_{n_i}^i}, [x']p'|_{\sigma_1^i}, .., [x']p'|_{\sigma_{n_i}^i})r'$, applying the ith sequence of reductions on both sides $\overset{\text{item (ii) and (***)}}{\Rightarrow} ([x]p)r \to^* p[([x]p|_{\sigma_{j_1}^i})r, \ldots, ([x]p|_{\sigma_{j_l}^i})r]_{\sigma_{j_1}^i,\ldots,\sigma_{j_l}^i} = p'[([x']p'|_{\sigma_{j_1}^i})r', \ldots, ([x']p'|_{\sigma_{j_l}^i})r']_{\sigma_{j_1}^i,\ldots,\sigma_{j_l}^i} \,{}^*\!\leftarrow ([x']p')r'$, then $([x]p|_{\sigma_{j_k}^i})r = ([x']p'|_{\sigma_{j_k}^i})r' \; \forall \sigma_{j_k}^i \in Par$, and by I.H. $p|_{\sigma_{j_k}^i}[r|x] = p'|_{\sigma_{j_k}^i}[r'|x'] \; \forall \sigma_{j_k}^i \in Par$, therefore $p[r|x] \overset{(**)}{=} p[p|_{\sigma_{j_1}^i}[r|x], \ldots, p|_{\sigma_{j_l}^i}[r|x]]_{\sigma_{j_1}^i,\ldots,\sigma_{j_l}^i} = p'[p'|_{\sigma_{j_1}^i}[r'|x'], \ldots, p'|_{\sigma_{j_l}^i}[r'|x']]_{\sigma_{j_1}^i,\ldots,\sigma_{j_l}^i} \overset{(**)}{=} p'[r'|x']$

The recursive rules is due to construction of $p[r|x]$ in the proof. □

With the same example of the first paragraph of this subsection, it can be seen that $[x]_A$ with $A \in \{HS, SH, BD\}$, does not satisfy the third condition of the previous theorem. However, $[x]_A$ with $A \in \{fab, H, abf, SH_{-\eta}, BD_{-\eta}\}$ satisfies all three, but to prove this in $BD_{-\eta}$, it is necessary to define $\mathscr{K}$ based on another combinatory logic.

Definition 4.11 A number is defined for each $\tau \in \mathscr{K}$ which is called *order of* τ *[7]*. The order of Φ_α is defined as $\#\alpha$, which is the number of b, c and K that occur in α.

Definition 4.12 The following super combinators are defined: $B^n := \lambda x_0, \ldots, x_{n+1}.x_1(x_0x_2 \ldots x_{n+1})$, $C^n := \lambda x_0, \ldots, x_{n+1}.(x_0x_2 \ldots x_{n+1})x_1$, $S^{n,m} := \lambda x_0, .., x_{n+m+2}.(x_0x_2 \ldots x_{n+1}x_{n+m+2})(x_1x_{n+2} \ldots x_{n+m+2})$. $\mathscr{K}$ can be defined in $CL(Var, \Sigma_{Z' \cup \{I,K\} \cup \{B^n, C^m, S^{i,j}\}_{\substack{i,j \geq 0 \\ n,m>0}}})$ as follows: $\Phi_\epsilon := I$, $\Phi_K := K$ and for $\alpha_1, \alpha_2 \in \mathscr{A}$ you have to $\Phi_{b\alpha_1} := B^{\#b\alpha_1}\Phi_{\alpha_1}$, $\Phi_{c\alpha_1} := C^{\#c\alpha_1}\Phi_{\alpha_1}$ and $\Phi_{(\alpha_1,\alpha_2)} := S^{\#\alpha_1,\#\alpha_2}\Phi_{\alpha_1}\Phi_{\alpha_2}$.

For the following theorem, the rewrite relation used for $BD_{-\eta}$ will be that of

$CL(Var, \Sigma_{Z' \cup \{I,K\} \cup \{B^n, C^m, S^{i,j}\}_{\substack{i,j \geq 0 \\ n,m>0}}})$.

Theorem 4.13 *$[x]_A$ with $A \in \{fab, H, abf, SH_{-\eta}, BD_{-\eta}\}$ satisfies the three conditions of the Theorem 4.10, furthermore $[x]_{BD_{-\eta}}p$ is of the form $|\Phi_\alpha|w$ (w could be ϵ), where w is formed by $\#\alpha$ arguments. The fuction κ of Theorem 4.10 for $BD_{-\eta}$ is computed as follows:*

- $\kappa(\Phi_\epsilon q) := q$,
- $\kappa(\Phi_K pq) := p$,
- $\kappa(\Phi_{c\alpha} p p_1 \dots p_{\#\alpha+1}) := \kappa(\Phi_\alpha p_1 \dots p_{\#\alpha+1})p$,
- $\kappa(\Phi_{b\alpha} p p_1 \dots p_{\#\alpha+1}) := p\kappa(\Phi_\alpha p_1 \dots p_{\#\alpha+1})$,
- $\kappa(\Phi_{(\alpha_1,\alpha_2)} p_1 \dots p_{\#\alpha_1} q_1 \dots q_{\#\alpha_2} t) := \kappa(\Phi_{\alpha_1} p_1 \dots p_{\#\alpha_1} t)\kappa(\Phi_{\alpha_2} q_1 \dots q_{\#\alpha_2} t)$

for fab, abf is computed as follows:

- $\kappa(Iq) := q$,
- $\kappa(Kpq) := p$,
- $\kappa(Sp_1p_2r) := \kappa(p_1r)\kappa(p_2r)$

for H is computed like fab replacing $\kappa(Iq) := q$ by $\kappa(SKKq) := q$ and for $SH_{-\eta}$ is computed as follows:

- $\kappa(Iq) := q$,
- $\kappa(Kpq) := p$,
- $\kappa(Cp_1p_2p_3) := \kappa(p_1p_3)p_2$,
- $\kappa(Bp_1p_2p_3) := p_1\kappa(p_2p_3)$,
- $\kappa(Sp_1p_2r) := \kappa(p_1r)\kappa(p_2r)$

Proof. Is easy to prove by structural induction that $[x]_{BD_{-\eta}}p = |\Phi_\alpha|w = \Phi_\alpha p_1..p_{\#\alpha}$. Now it is proved that the third condition is true.

The functions t_i of these bracket abstractions are of the form $|C_i|w$ where C_i is a combinator such that $C_i \neq C_j$ if $i \neq j$. therefore, if $(x, q) \in A_i$ and $[x]q = [x']r$, then $[x']r$ must be an image of t_i, since otherwise $[x]q = |C_i|w' \neq |C_j|w = [x']r$. This implies that $(x', r) \in A_i$.

Now it is proved that the first condition is true:

Case 1: $t_i(p) = I = \Phi_\epsilon$ or $t_i(p) = SKK$

In this case $(x, p) \in A_1$ and $p = x$, therefore $t_i(p)q = SKKq \to Kq(Kq) \to q = x[q|x] = p[q|x]$ if $t_i(p) = SKK$ or $t_i(p)q = Iq \to q = x[q|x] = p[q|x]$ otherwise. Thus it is defined $\kappa(SKKq) = q$ and $\kappa(\Phi_\epsilon q) = \kappa(Iq) = q$. Note that since $[x]_{BD_{-\eta}}q = |\Phi_\alpha|w$, then $t_i(p|_1, p|_2, [x]_{BD_{-\eta}}p|_1, [x]_{BD_{-\eta}}p|_2) \neq \Phi_\epsilon$.

Case 2: $t_i(p) = Kp$ $(K = \Phi_K)$

In this case $(x, p) \in A_2$ and $x \notin p$, therefore $t_i(p)q = Kpq \to p \overset{x \notin p}{=} p[q|x]$. Thus it is defined $\kappa(\Phi_K pq) = \kappa(Kpq) = p$

Now it is proved that the second condition is true:

Case 1.1: $t_i(p|_1, p|_2, [x]p|_1, [x]p|_2) = C([x]p|_1)p|_2$ or
Case 1.2: $t_i(p|_1, p|_2, [x]p|_1, [x]p|_2) = |\Phi_{c\alpha}|(p|_2 w)$
In both cases $(x,p) \in A_3$ and $x \notin p|_2$, therefore $t_i(p|_1, p|_2, [x]p|_1, [x]p|_2)r = C([x]p|_1)p|_2 r \to ([x]p|_1)rp|_2 = p[([x]p|_1)r]_1$ for case 1.1. For the other case,

$t_i(p|_1, p|_2, [x]p|_1, [x]p|_2)r = (|\Phi_{c\alpha}|(p|_2 w))r = (|\Phi_{c\alpha}p|_2|w)r = (|C^{\#c\alpha}\Phi_\alpha p|_2|w)r = C^{\#c\alpha}\Phi_\alpha p|_2 p_1 \dots p_{\#\alpha} r \to (\Phi_\alpha p_1 \dots p_{\#\alpha} r)p|_2 = ([x]p|_1)rp|_2 = p[([x]p|_1)r]_1$.

Note that 1 is a position belonging to the set of positions $\{1,2\}$, which are the ones used for $i = 3$. All occurrences of x in p are found in $p|_1$. In both cases, this reduction is independent of who (x,p) is in A_3, and the definitions $\kappa(Cp_1p_2p_3) := \kappa(p_1p_3)p_2$ and $\kappa(\Phi_{c\alpha}p_0p_1..p_{\#\alpha}r) := \kappa(\Phi_\alpha p_1..p_{\#\alpha}r)p_0$ satisfies $\kappa(([x]p)r) = p[\kappa(([x]p|_1)r)]_1$ as in Theorem 4.10.

Case 2.1: $t_i(p|_1, p|_2, [x]p|_1, [x]p|_2) = Bp|_1[x]p|_2$ or
Case 2.2: $t_i(p|_1, p|_2, [x]p|_1, [x]p|_2) = |\Phi_{b\alpha}|(p|_1 w)$
In both cases $(x,p) \in A_4$ and $x \notin p|_1$, therefore $t_i(p|_1, p|_2, [x]p|_1, [x]p|_2)r = Bp|_1([x]p|_2)r \to p|_1(([x]p|_2)r) = p[([x]p|_2)r]_2$ for case 2.1. For the other case,

$t_i(p|_1, p|_2, [x]p|_1, [x]p|_2)r = (|\Phi_{b\alpha}|(p|_1 w))r = (|\Phi_{b\alpha}p|_1|w)r = (|B^{\#b\alpha}\Phi_\alpha p|_1|w)r = B^{\#b\alpha}\Phi_\alpha p|_1 p_1 \dots p_{\#\alpha} r \to p|_1(\Phi_\alpha p_1 \dots p_{\#\alpha} r) = p|_1(([x]p|_2)r) = p[([x]p|_2)r]_2$.

Note that 2 is a position belonging to the set of positions $\{1,2\}$, which are the ones used for $i = 4$. All occurrences of x in p are found in $p|_2$. In both cases, this reduction is independent of who (x,p) is in A_4, and the definitions $\kappa(Bp_1p_2p_3) := p_1\kappa(p_2p_3)$ and $\kappa(\Phi_{b\alpha}p_0p_1..p_{\#\alpha}r) := p_0\kappa(\Phi_\alpha p_1..p_{\#\alpha}r)$ satisfies $\kappa(([x]p)r) = p[\kappa(([x]p|_2)r)]_2$ as in Theorem 4.10.

Case 3.1: $t_i(p|_1, p|_2, [x]p|_1, [x]p|_2) = S([x]p|_1)[x]p|_2$ or
Case 3.2: $t_i(p|_1, p|_2, [x]p|_1, [x]p|_2) = ||\Phi_{(\alpha_1,\alpha_2)}|w_1|w_2$
$t_i(p|_1, p|_2, [x]p|_1, [x]p|_2)r = S([x]p|_1)([x]p|_2)r \to (([x]p|_1)r)(([x]p|_2)r) = p[([x]p|_1)r, ([x]p|_2)r]_{1,2}$ for case 3.1. For the other case,

$t_i(p|_1, p|_2, [x]p|_1, [x]p|_2)r = (||\Phi_{(\alpha_1,\alpha_2)}|w_1|w_2)r = (||S^{\alpha_1,\alpha_2}\Phi_{\alpha_1}\Phi_{\alpha_2}|w_1|w_2)r \to (|\Phi_{\alpha_1}|w_1)r((|\Phi_{\alpha_2}|w_2)r) = (([x]p|_1)r)(([x]p|_2)r) = p[([x]p|_1)r, ([x]p|_2)r]_{1,2}$.

Note that 1 and 2 are positions belonging to $\{1,2\}$, which are the ones used for $i = 5$ (in $SH_{-\eta}$ and $BD_{-\eta}$) or $i = 3$ (for H, fab, abf). All occurrences of x in p are found in $p|_1$ and $p|_2$. In both cases, this reduction is independent of who (x,p) is, and the definitions $\kappa(Sp_1p_2r) := \kappa(p_1r)\kappa(p_2r)$ and $\kappa(\Phi_{(\alpha_1,\alpha_2)}p_1..p_{\#\alpha_1}q_1..q_{\#\alpha_2}r) := \kappa(\Phi_{\alpha_1}p_1 \dots p_{\#\alpha_1}r)\kappa(\Phi_{\alpha_2}q_1 \dots q_{\#\alpha_2}r)$ satisfies $\kappa(([x]p)r) = p[\kappa(([x]p|_1)r), \kappa(([x]p|_2)r)]_{1,2}$ as in Theorem 4.10.

□

Theorem 4.14 *If* $[x]$ *fits the general recursion scheme and satisfies* **BA** *then* $FreeVar([x]p) = FreeVar(p) \setminus \{x\}$

Proof. **BA** implies $([x]p)x \to^* p[x|x] = p$. Since in a contractum of a redex, subterms that are not in the redex cannot appear, but some of them may be absent, then $FreeVar(p) \subseteq FreeVar(([x]p)x) = FreeVar([x]p) \cup \{x\}$, therefore

$FreeVar(p) \setminus \{x\} \subseteq FreeVar([x]p)$. On the other hand, since per Lemma 4.1 $FreeVar([x]p) \subseteq FreeVar(p)$, but $x \notin [x]p$, then $FreeVar([x]p) \subseteq FreeVar(p) \setminus \{x\}$. □

Theorem 4.15 *If $t \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$ and $< t >$ is computed with a bracket abstraction that fits the general recursion scheme and satisfies* **BA**, *then $FreeVar(t) = FreeVar(< t >)$*

Proof. By structural induction on t and Theorem 4.14. □

4.4 Strong Reduction Property **SBA**

For Broda and Damas Combinators in [7], a rewrite relation different from that of Theorem 4.13 is used. This definition is as follows:

Definition 4.16 The following redex and contractum are defined for $CL(Var, \Sigma_{Z' \cup \mathscr{K}})$

- $\Phi_\alpha p_1 \dots p_{\#\alpha+1} \to_0 \kappa(\Phi_\alpha p_1 \dots p_{\#\alpha+1})$

Where the κ function is defined as in Theorem 4.13. The relation $\to$ in $CL(Var, \Sigma_{Z' \cup \mathscr{K}})$, is the smallest one that contains $\to_0$ and pass to the context

Remark 4.17 In [7] it is shown that **SBA** is almost true, precisely that $([x]_{BD}p)t \to^{0,1} p[t|x]$, however Theorem 4.13 has as a corollary, that for $[x]_{BD_{-\eta}}$, **SBA** is true using Definition 4.16 (this is the original definition in [7]).

Corollary 4.18 (SBA) *If $x \in Var$, $p, t \in CL(Var, \Sigma_Z)$, and using $\to$ of the Definition 4.16, then $([x]_{BD_{-\eta}}p)t \to p[t|x]$*

*4.5 Injectivity of $<>_\Lambda$ (***I***)*

Not all translations are injective, for example using the $A \in \{HS, SH, BD\}$ algorithms we have to $< (\lambda x.yx)z >_A = ([x]_A < yx >_A) < z >_A = ([x]_A yx)z = yz$, but on the other hand $< yz >_A = < y >_A < z >_A = yz$. Since $(\lambda x.yx)z$ and yz belong to different classes of Λ, then $<>_\Lambda$ computed with the indicated algorithms, is not injective.

From the previous example can be inferred that the function $<>_\Lambda$ will never be injective, if the eta rule is used to compute $[x]$. The following theorem, states which are the conditions that must be assumed, for property **I** to be fulfilled.

Theorem 4.19 (I) *If $[x]$ fits the general recursion scheme, $[x]r \notin Var \cup Z'$, $[x]$ satisfies the three conditions of theorem 4.10, and $[x] < p >$ is not unifiable with $< q >< t >$ then:*

$$\text{If } p, q \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}}) \text{ and } < p > = < q > \text{ then } p \equiv_\alpha q$$

Proof. By structural induction.
If $p \in Var \cup Z'$

$p =< p >=< q >$ since $[x]r \notin Var \cup Z'$ and if $q = q't$ then $< q >=< q' >< t > \notin Var$, then $q \in Var$

$\Rightarrow$

$p =< q >= q \Rightarrow p \equiv_\alpha q$

If $p = q't$

$< q' >< t >=< p >=< q >$, since if $q \in Var$ then $< q > \in Var$, and $[x] < q'' >$ is not unifiable with $< q' >< t >$, then $q = q''t'$. Therefore

$< p >=< q' >< t >=< q >=< q'' >< t' >\Rightarrow$

$< q' >=< q'' >$ and $< t >=< t' > \overset{I.H.}{\Rightarrow} q' \equiv_\alpha q''$ and $t \equiv_\alpha t' \Rightarrow p = q't \equiv_\alpha q''t' = q$

If $p = \lambda x.q'$

$[x] < q' >=< p >=< q >$, since $[x] < q' > \notin Var \cup Z'$ and $[x] < q' >$ is not unifiable with

$< q'' >< t' >$, then $q = \lambda x'.t$. Therefore

$[x] < q' >=< q >= [x'] < t > \overset{Var \ni x_c \ fresh \ in \ q't}{\Rightarrow} ([x] < q' >)x_c = ([x'] < t >)x_c$

κ *is a function*

$\Rightarrow$

$\kappa(([x] < q' >)x_c) = \kappa(([x'] < t >)x_c) \overset{def\ of\ \kappa}{\Rightarrow} < q' > [x_c|x] =< t > [x_c|x']$

Lema 4.5 x_c *fresh*

$\Rightarrow$

$< q' < x_c|x >>=< t < x_c|x' >> \overset{I.H.}{\Rightarrow} q' < x_c|x > \equiv_\alpha t < x_c|x' >$

Since x_c was any fresh variable, then it can be generalized, obtaining $(\forall x_c \in Var \setminus A)(q' < x_c|x > \equiv_\alpha t < x_c|x' >)$ where A is the set (finite) of the variables of q' and t. The latter is equivalent to $p \equiv_\alpha q$.

□

Corollary 4.20 *If $[x]$ fits the general recursion scheme, $[x]r \notin Var \cup Z'$, $[x]$ satisfies the three conditions of theorem 4.10, and $[x] < p >$ is not unifiable with $< q >< t >$ then:*

$$\textit{If } p, q \in \lambda\textit{-term}(Var, \Sigma_{\lambda_{Z'}}) \textit{ then } < p >=< q > \iff p \equiv_\alpha q$$

Now a recursive algorithm is defined, to compute the inverse function $<>^{-1}$: $Img(<>) \to \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$.

- $< p >^{-1} = p$ if $p \in Var \cup Z'$,
- $< p >^{-1} = \lambda x_c. < \kappa(px_c) >^{-1}$ if p is an image of $[x]$ for some x and where x_c is a fresh variable in p,
- $< pq >^{-1} =< p >^{-1} < q >^{-1}$

If there is ambiguity about what rule to take, the one listed first will be used.

Theorem 4.21 *If $[x]$ fits the general recursion scheme, $[x]r \notin Var \cup Z'$, $[x]$ satisfies the three conditions of theorem 4.10, $[x] < p >$ is not unifiable with $< q >< t >$ then:*

$$\textit{If } p \in \lambda\textit{-term}(Var, \Sigma_{\lambda_{Z'}}) \textit{ then } << p >>^{-1} \in \lambda\textit{-term}(Var, \Sigma_{\lambda_{Z'}}) \textit{ and } << p >>^{-1} \equiv_\alpha p$$

Proof. Note that if $u \equiv_\alpha u'$ and $y \notin BoundVar(uu')$ then $u < y|x > \equiv_\alpha u' < y|x >$ $(***)$
The proof is by structural induction over p.
If $p \in Var \cup Z'$

$<< p >>^{-1} \overset{def}{=} < p >^{-1} \overset{def}{=} p \equiv_\alpha p$ and
$<< p >>^{-1} = p \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$

If $p = qt$ with $q, t \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$

Since $< qt > = < q >< t >$ is not unifiable with $[x] < p' >$, then
$<< q >< t >>^{-1} = << q >>^{-1} << t >>^{-1}$. Since by **I.H.**
$<< q >>^{-1} \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$, $<< q >>^{-1} \equiv_\alpha q$, $<< t >>^{-1} \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$ and $<< t >>^{-1} \equiv_\alpha t$, then by definition of $\equiv_\alpha$, is fulfilled that $<< p >>^{-1} = << q >>^{-1} << t >>^{-1} \equiv_\alpha qt = p$ and
$<< p >>^{-1} = << q >>^{-1} << t >>^{-1} \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$.

If $p = \lambda x.q$ with $x \in Var$ and $q \in \lambda\text{-}term(Var, \Sigma_{\lambda_{Z'}})$

$<< p >>^{-1} = << \lambda x.q >>^{-1} = < [x] < q >>^{-1} \overset{[x]<q>\notin Var \cup Z' \text{ and definition of } <>^{-1}}{=}$

$\lambda x_c. < \kappa(([x] < q >)x_c) >^{-1} \overset{Theorem\ 4.10}{=} \lambda x_c. << q > [x_c|x] >^{-1} \overset{q' \equiv_\alpha q \ \wedge\ x_c \notin BoundVar(q') \ \wedge\ Theorem\ 4.6}{=}$
$\lambda x_c. << q' > [x_c|x] >^{-1}$.

It is now proved by cases that $< q' > [x_c|x] = < q' < x_c|x >>$.

If $x_c = x$ then $< q' > [x_c|x] = < q' > [x|x] = < q' > = < q' < x|x >> = < q' < x_c|x >>$.

If $x_c \neq x$ then, since x_c is fresh in $[x] < q >$, it is necessary that $x_c \notin FreeVar([x] < q >) \overset{Theorem\ 4.14}{=} FreeVar(< q >) \setminus \{x\} \overset{Theorem\ 4.15}{=} FreeVar(q) \setminus \{x\} = FreeVar(q') \setminus \{x\}$, so $x_c \notin FreeVar(q')$, in addition q' was chosen such that $x_c \notin BoundVar(q')$ and therefore $x_c \notin Var(q')$. Using Lemma 4.5 we have to $< q' > [x_c|x] = < q' < x_c|x >>$.

Thus $<< p >>^{-1} = \lambda x_c. << q' > [x_c|x] >^{-1} = \lambda x_c. << q' < x_c|x >>>^{-1}$

On the other hand defining $A := BoundVar(<< q' < x_c|x >>>^{-1} (q' < x_c|x >)) \cup BoundVar(qq')$ and taking $y \in Var \setminus A$, is fulfilled that:

$<< q' < x_c|x >>>^{-1} < y|x_c >$
$\overset{y \notin BoundVar(<<q'<x_c|x>>>^{-1}(q'<x_c|x>)),\ (****) \text{ and } I.H.}{\equiv_\alpha}$
$(q' < x_c|x >) < y|x_c >$
$\overset{x_c \notin BoundVar(q') \text{ and } x_c \notin FreeVar(q') \setminus \{x\}}{=}$
$q' < y|x >$
$\overset{y \notin BoundVar(qq') \text{ and } (****)}{\equiv_\alpha}$
$q < y|x >$

Since y was any variable in $Var \setminus A$, then it can be generalized, obtaining $(\forall y \in Var \setminus A)(<< q' < x_c|x >>>^{-1} < y|x_c > \equiv_\alpha q < y|x >)$ where A is a finite

set. The latter is equivalent to:

$$<< p >>^{-1} = \lambda x_c.\ << q' < x_c | x >>>^{-1} \equiv_\alpha \lambda x.q = p$$

□

References

[1] F. Baader, T. Nipkow, *Term Rewriting and All That*,Cambridge University Press, Cambridge, 1998.

[2] H. Barendregt, *The Lamba calculus, its syntax and semantics*,North-Holland, Amsterdam, 1981.

[3] Jean-Louis Krivine, *Lambda-calcul: types et modèles.*,Masson Editores, Paris, 1990.

[4] M. Schönfinkel, *Über die Bausteine der mathematischen Logik*,Mathematische Annalen, Vol 92 (1924), pp. 305-316.

[5] H. B. Curry, Robert Feys and William Craig, *Combinatory Logic*,volume 1. North-Holland, 1958.

[6] J.R.Hindley, B.Lercher y J.P.Seldin, *Introduction to Combinatory Logic.*,Cambridge University Press, Cambridge, 1972.

[7] S. Broda y L. Damas, *Compact Bracket Abstraction in Combinatory Logic*,The Journal of Symbolic Logic, Vol 62 (1997), pp. 729-740.

[8] Aledania: Lambda Calculus Evaluator. Avalaible http://stilgar.ldc.usb.ve/Aledania

[9] CalcLogic: Calculative Logic Asistant. Avalaible http://stilgar.ldc.usb.ve/CalcLogic

[10] M. S. Joy, V. J. Rayward-Smith and F. W. Burton. *Efficient combinator code.*Computer Languages, 10(3/4):211-224, 1985.

[11] M. S. Joy. *On the efficient implementation of combinators as an object code for functional programs.*PhD thesis, University of East Anglia, 1984.

[12] D. A. Turner, *Another algorithm for bracket abstraction.* The Journal of Symbolic Logic, 44:267-270,1979.

[13] D. A. Turner, *A new implementation technique for applicative languages.* Software-Practice and Experience, 9:31-49, 1979.

[14] T. Hikita, *On the average size of Turner's translation to combinator programs.* Journal of Information Processing, 7(3):164-169, 1984.

[15] L. Czajka, *On the equivalence off different presentations of Turner's bracket abstraction algorithm.* CoRR, abs/1510.03794, 2015.

[16] M. W. Bunder, *Some improvements to Turner's algorithm for bracket abstraction.* Journal of Symbolic Logic, 55(2):656-669, 1990.

[17] S. L. P. Jones, *The Implementation of Functional Programming Languages.* Prentice-Hall, 19877.

[18] R. J. M. Hughes, *Supercombinators: A new implementation method for applicative languages.* In LISP and Functional Programming. ACM Press, 1982.

[19] J. R. Kennaway and M. R. Sleep, *Variable abstraction in $O(n \log n)$ space.* Information Processing Letters, 24:343-349, 1987.

[20] R. Kennaway and M. R. Sleep, *Director strings as combinators.* ACM Trans. Program. Lang. Syst, 10(4):602-626, 1988.

[21] D. Stevens, *Variable substitution with iconic combinators.* In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, Mathematical Foundations of Computer Science, volume 711 of Lecture Notes in Computer Science, pages 724-733, Berlin, 1993. Springer-Verlag.

[22] A. R. Diller, *Efficent Bracket Abstraction Using Iconic Representations for Combinators.* Research Report, School of Computer Science, University of Birmingham, CSR-11-05, 2011.

[23] F. Flaviani, *Estudio comparativo de operadores de abstracción para la lógica combinatoria.* Universidad Simón Bolívar, Caracas, 2011.

[24] F. Flaviani, *Algoritmo de evaluación del λ cálculo basado en propiedades biyectivas de operadores de abstracción*. Universidad Simón Bolívar, Caracas, 2014.

[25] J. Carette, R. O'Connor, *Theory Presentation Combinators*. International Conf. on Intelligent Computer Mathematics, Lecture Notes in Computer Science, 2012.

[26] J. Carette, R. O'Connor, *Theory Presentation Combinators*. preprint of Journal of Formal Reasoning, 2018.

[27] L. Czajka, *Improving Automation in Interactive Theorem Provers by Efficient Encoding of Lambda-Abstractions*.Proc. ot the 5th ACM SIGPLAN Conf. on Certified Programs and Proofs, St. Petersburg, USA, 2016.

[28] M. Bunder, *Expedited Broda-Damas bracket abstraction*. Journal of Symbolic Logic, 65(4):1850-1857, 2000.