Completeness and Decidability of de Bruijn Substitution Algebra in Coq

Steven Schäfer Gert Smolka Tobias Tebbi
Saarland University
{schaefer,smolka,ttebbi}@ps.uni-saarland.de

Abstract

We consider a two-sorted algebra over de Bruijn terms and de Bruijn substitutions equipped with the constants and operations from Abadi et al.'s σ-calculus. We consider expressions with term variables and substitution variables and show that the semantic equivalence obtained with the algebra coincides with the axiomatic equivalence obtained with finitely many axioms based on the σ-calculus. We prove this result with an informative decision algorithm for axiomatic equivalence, which in the negative case returns a variable assignment separating the given expressions in the algebra. The entire development is formalized in Coq.

Categories and Subject Descriptors F.4.1 [*MATHEMATICAL LOGIC AND FORMAL LANGUAGES*]: Mathematical Logic—Lambda calculus and related systems

Keywords De Bruijn Terms, Explicit Substitutions, Finite Axiomatization, Decision Procedures, Algebra, Completeness, Coq

1. Introduction

We consider de Bruijn terms and substitution [5]. De Bruijn terms are syntactic trees $M ::= n \mid MM \mid \lambda M$ built over numbers n. The numbers serve as argument references for arguments introduced by the constructor λ. Substitutions are total functions mapping numbers to terms. The instantiation operation $M[f]$ instantiates the free (i.e., dangling) argument references of a term M with the terms provided by a substitution f such that substitution in the λ-calculus is modeled. See Abadi et al. [1] for a detailed explanation.

Two important operations on substitutions identified by Abadi et al. [1] are cons $M \cdot f$ and composition $f \circ g$. These operations are characterized by the identities $0[M \cdot f] = M$, $(n+1)[M \cdot f] = n[f]$, and $M[f \circ g] = M[f][g]$.

Two distinguished substitutions are the identity substitution $I = \lambda n.n$ and the shift substitution $S = \lambda n.\, n+1$. The identity substitution I can be expressed as $I = 0 \cdot S$.

We are interested in the two-sorted first-order algebra obtained with de Bruijn terms and de Bruijn substitutions and the operations 0, MN, λM, $M[f]$, S, $M \cdot f$, and $f \circ g$. We call the algebra *de Bruijn algebra*. We show in this paper that equality in the de Bruijn algebra is decidable and finitely axiomatizable.

The de Bruijn algebra constitutes a canonical semantical model for the different variants of the σ-calculus [1, 3, 4, 6]. A σ-calculus provides a first-order rewriting system for the expressions of the de Bruijn algebra. Typically, the extension of a σ-calculus with a rule $(\lambda M)N \to M[N \cdot I]$ for β-reduction is of main interest, and the design goal is a ground confluent system. So far, σ-calculi have been considered as purely syntactic systems. This is surprising, given that the de Bruijn algebra is such an obvious semantic model.

Abadi et al.'s [1] original σ-calculus is a ground confluent and terminating rewriting system, which is sound but not complete for the de Bruijn algebra. For instance, the ground equation $0[S] \cdot (S \circ S) = S$ cannot be shown since both sides are normal.

Curien et al's [3] σ_{SP}-calculus is a terminating [4] and confluent rewriting system extending the original σ-calculus.[1] The equational theory of the σ_{SP}-calculus is deductively equivalent to our axiomatization of the de Bruijn algebra. Thus the rewriting system of the σ_{SP}-calculus provides a sound and complete decision procedure for equality in the de Bruijn algebra.

The motivation for the research presented in this paper came from a formal development of λ-calculus in Coq where the de Bruijn algebra appears at the lowest level of the formalization. Substitution lemmas needed for this development state identities for the de Bruijn algebra. For instance, the substitution lemma needed for proving that β-reduction is stable under substitution takes the form $M[N \cdot I][f] = M[0 \cdot (f \circ S)][N[f] \cdot I]$. After proving a few of these lemmas by hand the question for a systematic proof method came up. It turned out that we could prove all substitution lemmas by rewriting with the basic substitution lemmas forming the rewriting rules of the σ_{SP}-calculus. The completeness result of this paper together with the confluence and termination of the σ_{SP}-calculus establish the completeness of this proof method.

We do not use rewriting methods to obtain our completeness result. We obtain the completeness result with an informative decision algorithm that given two expressions decides whether they are axiomatically equivalent. In case the expressions are not axiomatically equivalent, the algorithm constructs a separating assignment for the expressions. Completeness of axiomatic equivalence for the de Bruijn algebra then follows from the correctness of the algorithm. The separating assignments constructed by the algorithm employ only substitutions denotable with ground expressions.

The decision algorithm is based on a carefully designed class of normal expressions. Every expression can be translated to an equivalent normal expression. Semantic equivalence of normal expressions can be characterized as equality modulo cons expansion of expressions describing substitutions.

[1] The σ_{SP}-calculus appears under different names in the literature: as σ_{SP}-calculus in [3], as σ'-calculus in [4], and simply as σ-calculus in [6].

CPP'15, January 12–14, 2015, Mumbai, India.

ACM 978-1-4503-3296-5/15/01…$15.00.
http://dx.doi.org/10.1145/2676724.2693163

It now suffices to construct an informative decision algorithm for normal expressions. The challenge here is the construction of the separating assignment in the negative case. We approach the problem with a proof system for apartness (the complement of equivalence). The basic decision algorithm now constructs a derivation certifying either the equivalence or the apartness of two expressions. The soundness of the apartness system is obtained with a separation algorithm that given an apartness derivation constructs a separating assignment. Since the separating assignment cannot be constructed in one go, the separation algorithm works with special environments describing sufficiently large sets of separating assignments.

The entire development of the paper is formalized in Coq with Ssreflect. In fact, the results have been developed hand in hand with the Coq formalization. The Coq development does not assume functional extensionality but employs equivalence of substitutions and setoid rewriting. Rewriting the development to Coq without Ssreflect would not be difficult. The Coq development accompanying the paper can be found at www.ps.uni-saarland.de/extras/cpp15.

The completeness result presented in this paper is new. So far, the σ-calculus has been studied as a rewriting system and has not been related to a semantic model. While it is obvious that the de Bruijn algebra is a canonical model of the σ-calculus, the completeness of the axioms of the σ-calculus for this model is not (keep in mind that there are variables ranging over functions).

The decidability result presented in this paper is not new since it can be obtained from the decidability result for the σ-calculus. However, our decision method is quite different from the established decision method for the σ-calculus, which is obtained with a confluent and terminating rewriting system. While the verification of our decision method is not difficult (even in Coq), a verification of the rewriting method is surprisingly complex since the existing termination proof [4] is far from straightforward. We did not succeed in simplifying this proof and think that a formalization with a proof assistant is a substantial enterprise.

2. Terms and Substitutions

We define a type $\mathbb{T}$ of *terms* inductively:

$$M, N \;::=\; n \mid MN \mid \lambda M \qquad (n \in \mathbb{N})$$

For the mathematical presentation we assume $\mathbb{N} \subseteq \mathbb{T}$. Terms of the form n, MN, and λM are called *indices*, *applications*, and *abstractions*, respectively.

A *substitution* is a function $\mathbb{N} \to \mathbb{T}$. A *renaming* is a substitution $\mathbb{N} \to \mathbb{N}$. Note that we do not require that renamings are bijective. We assume that equality of functions is extensional. The letters f, g, h will denote substitutions. A substitution can be seen as an infinite sequence $(M_0, M_1, M_2, \ldots)$ of terms. This view motivates the operations *head*, *tail*, and *cons*:

$$\begin{aligned} hd\ f &:= f0 \\ tl\ f &:= \lambda n.f(n+1) \\ M \cdot f &:= \lambda n.\ \mathsf{if}\ n = 0\ \mathsf{then}\ M\ \mathsf{else}\ f(n-1) \end{aligned}$$

We have $f = hd\ f \cdot tl\ f$ for every substitution f.

We introduce notation for the *identity* and the *shift* substitution:

$$\begin{aligned} I &:= \lambda n.n \\ S &:= \lambda n.n+1 \end{aligned}$$

Note that I and S are renamings satisfying $I = 0 \cdot S$.

Next we define the *instantiation* operation $M[f] \in \mathbb{T}$ (read M under f) that applies a substitution f to a term M. The definition requires care. We define instantiation together with a *composition* operation $f \circ g \in \mathbb{N} \to \mathbb{T}$ for substitutions f and g such that the following equations are satisfied.

$$\begin{aligned} n[f] &= fn \\ (MN)[f] &= (M[f])(N[f]) \\ (\lambda M)[f] &= \lambda(M[0 \cdot (f \circ S)]) \\ f \circ g &= \lambda n.\ (fn)[g] \end{aligned}$$

Given composition, the first three equations provide for a structurally recursive definition of instantiation. However, the definition of composition requires instantiation. We avoid mutual recursion and define composition and instantiation in five consecutive steps.

1. Define composition $f \circ g$ for renamings f and g. Instantiation is not needed.
2. Define instantiation $M[f]$ for renamings f by structural recursion on M.
3. Define composition $f \circ g$ for substitutions f and renamings g.
4. Define instantiation $M[f]$ for substitutions f.
5. Define composition $f \circ g$ for substitutions f and g.

The use of composition for the definition of instantiation is a matter of convenience. One can inline composition and define instantiation first for renamings and then for general substitutions (corresponding to steps (2) and (4)). One then defines composition based on instantiation (corresponding to step (5)). Such a two-level definition of instantiation is used by Adams [2].

Notationally, we give instantiation higher precedence than composition, and composition higher precedence than cons. Thus $M \cdot f \circ g$ stands for $M \cdot (f \circ g)$.

We have $hd\ f = 0[f]$ and $tl\ f = S \circ f$. Thus head and tail can be expressed with instantiation and composition.

We define *powers* f^n of substitutions as one would expect:

$$\begin{aligned} f^0 &:= I \\ f^{n+1} &:= f \circ f^n \end{aligned}$$

We have $hd(S^n) = n$, $tl(S^n) = S^{n+1}$, and $hd(S^n \circ f) = fn$.

We define *drops* ${}^nf$ of substitutions as follows:

$${}^nf \;:=\; \lambda k.\ f(k+n)$$

In the sequence view, the sequence ${}^nf$ is obtained from the sequence f by removing the first n elelements. We have ${}^0f = f$, ${}^1f = tl\ f$, and ${}^nf = S^n \circ f$.

Fact 1 *For every number n and every substitution f, the terms n, fn, $hd\ f$ and the substitutions I, $tl\ f$, f^n, ${}^nf$ can be expressed with just 0, S, cons, instantiation, composition, and f.*

Fact 2 *$S^{n+1} = S^n \circ S$ and ${}^nS = S^{n+1}$.*

PROOF By induction on n. ∎

A term M is *closed* if $M[f] = M$ for every substitution f.

A substitution f is *regular* if there are numbers $k, n \geq 0$ and terms $M_1, \ldots, M_k$ such that

$$f \;=\; M_1 \cdot \ldots \cdot M_k \cdot S^n$$

This representation of regular substitutions becomes unique if we require $n \neq M_k + 1$ in case M_k is an index. One can argue that regular substitutions suffice for term instantiation.

Fact 3 *I and S are regular substitutions. Moreover, regular substitutions are closed under cons and composition.*

3. Expressions and Semantic Equivalence

We consider *expressions* denoting terms and substitutions.

$$s, t, u \ ::= \ 0 \mid st \mid \lambda s \mid s[\sigma] \mid x$$
$$\sigma, \tau, \theta \ ::= \ S \mid s \cdot \sigma \mid \sigma \circ \tau \mid \xi$$

The expressions for terms (s, t, u) are formed with 0, term application, term abstraction, instantiation, and variables (x, y, z) ranging over terms. The expressions for substitutions (σ, τ, θ) are formed with S, cons, composition, and variables (ξ) ranging over substitutions. A *ground* expression is an expression not containing variables.

As before, we give instantiation higher precedence than composition, and composition higher precedence than cons.

Denotation of expressions is defined as one would expect. An *assignment* is a function mapping term variables to terms and substitution variables to substitutions. Every assignment yields two functions, one mapping term expressions to terms, and one mapping substitution expressions to substitutions. We use $\hat{\alpha}s$ and $\hat{\alpha}\sigma$ to denote the term and the substitution obtained from a term expression s and a substitutions expression σ with an assignment α. An assignment is *regular* if it maps every substitution variable to a regular substitution.

Fact 4 (Denotation)

1. *Every term and every regular substitution can be denoted with a ground expression.*
2. *Every substitution expression denotes a regular substitution under a regular assignment.*

PROOF The first claim holds since we have $I = 0 \cdot S$ and $n = 0[S^n]$. The second claim follows from Fact 3. ∎

Semantic equivalence of expressions is defined as one would expect:

$$s \approx t \ := \ \forall \alpha.\, \hat{\alpha}s = \hat{\alpha}t$$
$$\sigma \approx \tau \ := \ \forall \alpha.\, \hat{\alpha}\sigma = \hat{\alpha}\tau$$

An assignment *separates* two expressions if the expressions denote different objects under the assignment. Two expressions cannot be both separable and semantically equivalent.

Expression can describe I, powers, drops, heads, tails, indices, and applications of substitutions to numbers. We introduce the following abbreviations:

$$\begin{aligned} I &:= 0 \cdot S & hd\ \sigma &:= 0[\sigma] & \\ \sigma^0 &:= I & tl\ \sigma &:= S \circ \sigma & \\ \sigma^{n+1} &:= \sigma \circ \sigma^n & n &:= 0[S^n] & \text{if } n \geq 1 \\ {}^n\sigma &:= S^n \circ \sigma & \sigma n &:= n[\sigma] & \end{aligned}$$

4. Axiomatic Equivalence

We define *axiomatic equivalence* of expressions by means of the least congruence relations $s \equiv t$ and $\sigma \equiv \tau$ satisfying the following axioms.

$$\begin{aligned} (st)[\sigma] &\equiv (s[\sigma])(t[\sigma]) & I \circ \sigma &\equiv \sigma \\ (\lambda s)[\sigma] &\equiv \lambda(s[0 \cdot \sigma \circ S]) & \sigma \circ I &\equiv \sigma \\ 0[s \cdot \sigma] &\equiv s & (\sigma \circ \tau) \circ \theta &\equiv \sigma \circ (\tau \circ \theta) \\ S \circ (s \cdot \sigma) &\equiv \sigma & (s \cdot \sigma) \circ \tau &\equiv s[\tau] \cdot \sigma \circ \tau \end{aligned}$$

As stated, there are infinitely many axioms. However, if we replace the metavariables s, t, σ, τ, and θ with object variables (e.g., x and ξ), we have a finite axiom system.

Our axioms are deductively equivalent to the rewrite rules of Curien et al.'s σ_{SP}-calculus [3] (up to the detail that I is primitive in the σ_{SP}-calculus). Given that we do not require a confluent and terminating rewriting system, there is considerable freedom in the choice of axioms.

Lemma 5 (Soundness) *Axiomatically equivalent expressions are semantically equivalent.*

Fact 6 *Here are some derivable axiomatic equivalences.*

$$\begin{aligned} s[I] &\equiv s & \sigma &\equiv hd\ \sigma \cdot tl\ \sigma \\ s[\sigma][\tau] &\equiv s[\sigma \circ \tau] & S^n &\equiv n \cdot S^{n+1} \\ & & {}^n\sigma &\equiv \sigma n \cdot {}^{n+1}\sigma \end{aligned}$$

PROOF

- $s[I] \equiv 0[s[I] \cdot I \circ I] \equiv 0[(s \cdot I) \circ I] \equiv 0[s \cdot I] \equiv s$
- $s[\sigma][\tau] \equiv 0[s[\sigma][\tau] \cdot I \circ \sigma \circ \tau] \equiv 0[(s \cdot I) \circ \sigma \circ \tau] \equiv 0[s[\sigma \circ \tau] \cdot I \circ (\sigma \circ \tau)] \equiv s[\sigma \circ \tau]$
- $hd\ \sigma \cdot tl\ \sigma = 0[\sigma] \cdot S \circ \sigma \equiv (0 \cdot S) \circ \sigma = I \circ \sigma \equiv \sigma$
- $n \cdot S^{n+1} \equiv 0[S^n] \cdot S^{n+1} = 0[S^n] \cdot S \circ S^n \equiv (0 \cdot S) \circ S^n = I \circ S^n \equiv S^n$
- $\sigma n \cdot {}^{n+1}\sigma \equiv n[\sigma] \cdot S^{n+1} \circ \sigma \equiv (n \cdot S^{n+1}) \circ \sigma \equiv S^n \circ \sigma \equiv {}^n\sigma$

∎

We will establish the decidability and the coincidence of semantic and axiomatic equivalence with an informative decision algorithm for axiomatic equivalence, which returns a separating assignment in case the expressions are not semantically equivalent.

5. Normal Expressions

We now define a class of normal expressions for which we will construct an informative decision algorithm. This will yield an informative decision algorithm for all expressions since every expression can be translated into an equivalent normal expression.

The basic idea behind normal term expressions is that ground normal expressions are verbatim descriptions of terms and regular substitutions. Here is the definition of *normal expressions*:

$$v \ ::= \ x \mid \xi n$$
$$s, t \ ::= \ n \mid st \mid \lambda s \mid v[\sigma]$$
$$\sigma, \tau \ ::= \ S^n \mid s \cdot \sigma \mid {}^n\xi \circ \sigma$$

Normal expressions are build over the types of numbers (letter n), term variables (letter x), and substitution variables (letter ξ). Instantiation and composition of normal expressions are restricted in that the left constituent must contain a variable, and variables can only appear in such positions.

Given the abbreviations stated at the end of Section 3, we may see normal expressions as ordinary expressions. However, we define normal expressions as a separate data type so that we obtain a suitable structural recursion scheme for normal expressions. The forms n, $v[\sigma]$, S^n, and ${}^n\xi \circ \sigma$ are obtained with dedicated constructors.

Since normal expressions employ the same variables as ordinary expressions, we can evaluate normal expressions under the same assignments we use for ordinary expressions. The *denotation* of a normal expression under an assignment α is defined as one would expect.

$$\begin{aligned} \hat{\alpha}x &:= \alpha x & & \\ \hat{\alpha}(\xi n) &:= (\alpha\xi)n & & \\ \hat{\alpha}n &:= n & \hat{\alpha}(S^n) &:= S^n \\ \hat{\alpha}(st) &:= (\hat{\alpha}s)(\hat{\alpha}t) & \hat{\alpha}(s \cdot \sigma) &:= \hat{\alpha}s \cdot \hat{\alpha}\sigma \\ \hat{\alpha}(\lambda s) &:= \lambda(\hat{\alpha}s) & \hat{\alpha}({}^n\xi \circ \sigma) &:= {}^n(\alpha\xi) \circ \hat{\alpha}\sigma \\ \hat{\alpha}(v[\sigma]) &:= (\hat{\alpha}v)[\hat{\alpha}\sigma] & & \end{aligned}$$

Semantic equivalence of normal expressions is defined as one would expect.

We define a function γ translating normal expressions into semantically equivalent ordinary expressions using the abbreviations for ordinary expressions stated at the end of Section 3.

Fact 7 *$\hat{\alpha}s = \hat{\alpha}(\gamma s)$ and $\hat{\alpha}\sigma = \hat{\alpha}(\gamma\sigma)$ for all normal expressions s and σ.*

PROOF By structural induction on s and σ. ∎

We define *axiomatic equivalence* of normal expressions by means of the least congruences $s \equiv t$ and $\sigma \equiv \tau$ satisfying the following equivalences.

$$\begin{aligned} S^n &\equiv n \cdot S^{n+1} \\ {}^n\xi \circ \sigma &\equiv (\xi n)[\sigma] \cdot {}^{n+1}\xi \circ \sigma \end{aligned}$$

Left-to-right rewriting with the axioms is called *cons expansion*, and right-to-left rewriting is called *cons reduction*. Note that every normal substitution expression is either a cons expression or can be expanded to a cons expression.

Fact 8 (Soundness) *If normal expressions are axiomatically equivalent, then they are semantically equivalent.*

Fact 9 *If s and t are normal expressions such that $s \equiv t$, then $\gamma s \equiv \gamma t$.*

PROOF By induction on the derivation of $s \equiv t$ and Fact 6. ∎

Since every normal substitution expression is equivalent to a normal cons expression, we can define functions from normal expressions to normal expressions that act as head, tail, drop, and apply operations:

$$\begin{aligned} H(S^n) &:= n & T(S^n) &:= S^{n+1} \\ H(s \cdot \sigma) &:= s & T(s \cdot \sigma) &:= \sigma \\ H({}^n\xi \circ \sigma) &:= \xi n[\sigma] & T({}^n\xi \circ \sigma) &:= {}^{n+1}\xi \circ \sigma \\ D0\sigma &:= \sigma & \sigma@n &:= H(Dn\sigma) \\ D(n+1)\sigma &:= T(Dn\sigma) \end{aligned}$$

Fact 10 *$\gamma(H\sigma) \equiv hd(\gamma\sigma)$, $\gamma(T\sigma) \equiv tl(\gamma\sigma)$, $\gamma(Dn\sigma) \equiv {}^n(\gamma\sigma)$, and $\gamma(\sigma@n) = (\gamma\sigma)n$ for all normal expressions σ and all numbers n.*

Fact 11 (Cons Expansion) *$\sigma \equiv H\sigma \cdot T\sigma$ for every normal expression σ.*

Note that $S^n \equiv n \cdot S^{n+1}$ and ${}^n\xi \circ \sigma \equiv \xi n[\sigma] \cdot {}^{n+1}\xi \circ \sigma$ are instances of the equivalence $\sigma \equiv H\sigma \cdot T\sigma$ (take $\sigma = S^n$ and $\sigma = {}^n\xi \circ \sigma$). Thus we could have taken $\sigma \equiv H\sigma \cdot T\sigma$ as the single axiom defining equivalence of normal expressions.

Fact 12 *$\hat{\alpha}(H\sigma) = hd(\hat{\alpha}\sigma)$, $\hat{\alpha}(T\sigma) = tl(\hat{\alpha}\sigma)$, $\hat{\alpha}(Dn\sigma) = {}^n(\hat{\alpha}\sigma)$, and $\hat{\alpha}(\sigma@n) = (\hat{\alpha}\sigma)n$ for all assignments α, normal expressions σ, and numbers n.*

6. Reduction to Normal Expressions

We now define a function η translating ordinary expressions into normal expressions. The correctness requirement for η is stated by Fact 13. We define η by structural recursion on ordinary expressions.

$$\begin{aligned} \eta 0 &:= 0 & \eta S &:= S^1 \\ \eta(st) &:= (\eta s)(\eta t) & \eta(s \cdot \sigma) &:= \eta s \cdot \eta\sigma \\ \eta(\lambda s) &:= \lambda(\eta s) & \eta(\sigma \circ \tau) &:= C(\eta\sigma)(\eta\tau) \\ \eta(s[\sigma]) &:= J(\eta s)(\eta\sigma) & \eta\xi &:= {}^0\xi \circ S^0 \\ \eta x &:= x[S^0] \end{aligned}$$

The definition assumes two functions J and C taking normal expressions to normal expressions and satisfying the equivalences stated by Fact 14.

Fact 13 *$\gamma(\eta s) \equiv s$ and $\gamma(\eta\sigma) \equiv \sigma$ for all ordinary expressions s and σ.*

PROOF Follows by induction on s and σ with Fact 14. ∎

We now come to the definitions of the functions J and C. Given two normal expressions, these functions compute normal expressions representing the instantiation and the composition of the two expression as made precise by Fact 14. The definition follows the stratified definition of instantiation and composition for terms and substitutions in Section 2. We obtain functions J and C satisfying the following equations.

$$\begin{aligned} Jn\sigma &= \sigma@n & C(S^n)\tau &= Dn\tau \\ J(st)\sigma &= (Js\sigma)(Jt\sigma) & C(s \cdot \sigma)\tau &= Js\tau \cdot C\sigma\tau \\ J(\lambda s)\sigma &= \lambda(Js(0 \cdot C\sigma S^1)) & C({}^n\xi \circ \sigma)\tau &= {}^n\xi \circ C\sigma\tau \\ J(v[\tau])\sigma &= v[C\tau\sigma] \end{aligned}$$

The stratified definition of J and C requires a type of renaming expressions and proceeds as follows.

1. Define *renaming expressions* as $\sigma ::= S^n \mid n \cdot \sigma$.
2. Define composition of renaming expressions by structural recursion.
3. Define instantiation with renaming expressions and composition of substitution expressions with renaming expressions by mutual structural recursion.
4. Define instantiation with substitution expressions and composition of substitution expressions by mutual structural recursion.

Fact 14 *$\gamma(Js\sigma) \equiv (\gamma s)[\gamma\sigma]$ and $\gamma(C\sigma\tau) \equiv \gamma\sigma \circ \gamma\tau$ for all normal expressions s, σ, and τ.*

7. Deciding Equivalence and Apartness

We are aiming at an informative decision algorithm for equivalence of normal expressions returning a separating assignment for non-equivalent expressions. We factor the complexity of the algorithm by introducing *apartness derivations*. An apartness derivation for two expressions is a data structure from which one can construct a separating assignment for the expressions. We will have two algorithms, one that decides equivalence and yields an apartness derivation in the negative case, and one that translates apartness derivations into separating assignments.

To argue the termination of the decision algorithm, we need the *size* of normal expressions:

$$\begin{aligned} |n| &= 1 & |S^n| &= 1 \\ |st| &= 1 + |s| + |t| & |s \cdot \sigma| &= 1 + |s| + |\sigma| \\ |\lambda s| &= 1 + |s| & |{}^n\xi \circ \sigma| &= 1 + |\sigma| \\ |v[\sigma]| &= 1 + |\sigma| \end{aligned}$$

Fact 15 *$|H\sigma| \le |\sigma|$, $|T\sigma| \le |\sigma|$, $|Dn\sigma| \le |\sigma|$, and $|\sigma@n| \le |\sigma|$. If σ is a cons expression, then $|H\sigma| < |\sigma|$, $|T\sigma| < |\sigma|$, and $|\sigma@n| < |\sigma|$.*

The algorithm considers two normal expressions axiomatically equivalent if they are equal up to cons expansion at positions where the other side is already a cons expression. This idea can be made precise with a *proof system for axiomatic equivalence* of normal expressions:

$$\frac{}{n \equiv n} \qquad \frac{s_1 \equiv s_2 \qquad t_1 \equiv t_2}{s_1 t_1 \equiv s_2 t_2} \qquad \frac{s \equiv t}{\lambda s \equiv \lambda t} \qquad \frac{\sigma \equiv \tau}{v[\sigma] \equiv v[\tau]}$$

$$\frac{}{S^n \equiv S^n} \qquad \frac{\sigma \equiv \tau}{{}^n\xi \circ \sigma \equiv {}^n\xi \circ \tau} \qquad \frac{H\sigma \equiv H\tau \qquad T\sigma \equiv T\tau}{\sigma \equiv \tau}$$

The last rule is restricted to the case where σ or τ is a cons expression.

Fact 16 (Soundness) *The proof system for axiomatic equivalence of normal expressions is sound.*

PROOF Follows with Fact 11. ∎

As it will turn out, the proof system is complete for axiomatic equivalence of normal expressions (see Corollary 31).

The proof system for axiomatic equivalence is algorithmic in that for every rule the premises are smaller than the conclusion, where the size of $s \equiv t$ is $|s| + |t|$ and of $\sigma \equiv \tau$ is $|\sigma| + |\tau|$.

Fact 17 (Inversion) *Let σ and τ be normal expressions such that neither σ nor τ is a cons expressions. Then one can obtain a derivation of $\sigma \equiv \tau$ from a derivation of $H\sigma \equiv H\tau$.*

PROOF By case analysis. Note that σ and τ must have the form S^n or ${}^n\xi \circ \sigma'$. We have $H(S^n) = n$ and $H({}^n\xi \circ \sigma') = \xi n[\sigma']$. Since $H\sigma \equiv H\tau$, we have either $H\sigma = n = H\tau$ or $H\sigma = \xi n[\sigma']$, $H\tau = \xi n[\tau']$, and $\sigma' \equiv \tau'$. The claim $\sigma \equiv \tau$ now follows with either the first or the second rule for the equivalence of normal substitution expressions. ∎

We define the *variant number* of a normal term expression as follows:

$$\delta n \;:=\; 3 + n \quad \delta(st) \;:=\; 0 \quad \delta(\lambda s) \;:=\; 1 \quad \delta(v[\sigma]) \;:=\; 2$$

We define a *proof system for apartness* of normal expressions as follows.

$$\frac{\delta s \neq \delta t}{s \# t} \qquad \frac{s_1 \# s_2 \;\text{ or }\; t_1 \# t_2}{s_1 t_1 \# s_2 t_2} \qquad \frac{s \# t}{\lambda s \# \lambda t}$$

$$\frac{v_1 \neq v_2}{v_1[\sigma] \# v_2[\tau]} \qquad \frac{\sigma @ n \# \tau @ n}{v[\sigma] \# v[\tau]} \qquad \frac{\sigma @ n \# \tau @ n}{\sigma \# \tau}$$

The apartness rules are complementary to the proof rules for axiomatic equivalence. At this point neither soundness nor completeness of the apartness rules are obvious. Eventually both properties will be shown (Corollary 30). The apartness rules are designed such that they facilitate the construction of separating assignments. In contrast to the equivalence system, there is no mutual recursion between term expressions and substitution expressions in the apartness system.

Theorem 18 (Decision Algorithm) *There is an algorithm that for two normal expressions constructs either an equivalence derivation or an apartness derivation.*

PROOF We define the *rank* of two expressions as follows: $\rho st = |s| + |t|$ and $\rho\sigma\tau = 1 + |\sigma| + |\tau|$. The algorithm recurses on the rank of the expressions and the correctness proof is by induction on this rank. The recursions and the uses of the inductive hypothesis can be validated with Fact 15.

1. Consider s and t. If $\delta s \neq \delta t$, the claim follows with the respective apartness rule. Otherwise, let $\delta s = \delta t$. If s and t are applications or abstractions, the claim follows with the inductive hypothesis. If s is an index, then $s = t$ and the claim follows. If $s = v_1[\sigma]$ and $t = v_2[\tau]$ and $v_1 \neq v_2$, the claim follows with the respective apartness rule. This leaves us with the case $s = v[\sigma]$ and $t = v[\tau]$. The claim follows with the inductive hypothesis for σ and τ.
2. Consider σ and τ. By the inductive hypothesis we have a derivation for either $H\sigma \equiv H\tau$ or $H\sigma \# H\tau$. In the second case, we have $\sigma \# \tau$ since $H\sigma = \sigma @ 0$ and $H\tau = \tau @ 0$. Otherwise we have a derivation for $H\sigma \equiv H\tau$. Case analysis.
   (a) Either σ or τ is a cons expression. Then we have a derivation for either $T\sigma \equiv T\tau$ or $T\sigma \# T\tau$ by the inductive hypothesis. The claim follows with Fact 11.
   (b) Neither σ nor τ is a cons expression. Then Fact 17 gives us a derivation for $\sigma \equiv \tau$. ∎

8. Separation

In this section, a *variable* is either a term variable x or a pair ξn of a substitution variable and a number. This view is anticipated in the definition of normal expressions. If we assign terms to all variables x and ξn, we obtain an assignment as defined in Section 3. Given a substitution variable ξ, the terms for the variables ξn define the substitution for ξ pointwise. The letter v will always range over variables.

Our goal is an algorithm that, given an apartness derivation for two normal expressions, by recursion on the apartness derivation constructs an assignment separating the expressions. For every apartness rule but

$$\frac{\sigma @ n \# \tau @ n}{v[\sigma] \# v[\tau]}$$

it is clear how a separating assignment for the conclusion can be obtained. For the above rule the situation is as follows. By recursion we obtain an assignment α and a number n sucht that $\hat\alpha\sigma n \neq \hat\alpha\tau n$. If n is free in $\hat\alpha v$, then α separates $v[\sigma]$ and $v[\tau]$ and we are done. However, if n is not free in $\hat\alpha v \neq n$, it is not clear how to get a separating assignment for $v[\sigma]$ and $v[\tau]$. We cannot simply change α on v since this may destroy the inequality $\hat\alpha\sigma n \neq \hat\alpha\tau n$.

It turns out that there is a technique making it possible to change α on v without losing the inequality $\hat\alpha\sigma n \neq \hat\alpha\tau n$. For this we work with lists of terms and define a mapping that represents term lists as terms:

$$\overline{[]} \;:=\; \lambda 0$$
$$\overline{M :: A} \;:=\; M\,\overline{A}$$

For instance, we have $\overline{[M_1; M_2; M_3]} = M_1(M_2(M_3(\lambda 0)))$. The letters A and B will always range over term lists.

Fact 19 (Injectivity) *$\overline{A}$ and $\overline{B}$ are different terms if A and B are different term lists.*

An *environment* is a function from variables to term lists that maps all but a finite number of variables to the empty list. We use the letters φ and ψ to denote environments. Every environment φ represents an assignment $\overline{\varphi}$ as follows:

$$\overline{\varphi} x \;:=\; \overline{\varphi x}$$
$$\overline{\varphi}\,\xi \;:=\; \lambda n.\ \mathsf{if}\ \varphi(\xi n) = []\ \mathsf{then}\ n\ \mathsf{else}\ \overline{\varphi(\xi n)}$$

Fact 20 *If φ is an environment, then $\overline{\varphi}$ is a regular assignment.*

Concatenation of environments is defined pointwise:

$$\varphi +\!\!+ \psi \;:=\; \lambda v.\ \varphi v +\!\!+ \psi v$$

We now come to the central definition of this section. An environment φ *separates* two expressions

- s and t if $\widehat{\overline{\varphi + \psi}}(s) \neq \widehat{\overline{\varphi + \psi}}(t)$ for every environment ψ.
- σ and τ if there is a number n such that φ separates $\sigma@n$ and $\tau@n$.

We say that two expressions are *environment separable* if there is an environment separating them. The important point about a separating environment is that it remains separating under extension by concatenation.

Fact 21 (Extensibility) *If two expressions are separated by an environment φ, they are separated by all concatenation environments $\varphi + \psi$.*

Fact 22 *If two expressions are separated by an environment φ, they are separated by the regular assignment $\overline{\varphi}$.*

We now come to the construction of separating environments for apartness derivations. The construction is by structural recursion on the apartness derivation. Here is an example.

$$\dfrac{\dfrac{0 \# 1}{x[0 \cdot I] \# x[I]}\ n = 1}{x[x[0 \cdot I] \cdot S] \# x[x[I] \cdot S]}\ n = 0 \qquad \begin{array}{c} \{\} \\ \{x := [1]\} \\ \{x := [1;0]\} \end{array}$$

The apartness derivation appears on the left and the accompanying separating environments appear on the right. The following facts and lemmas prepare the proof of the Separation Theorem 29.

Fact 23 $\widehat{\overline{\varphi}}(v) = \overline{\varphi v}$ *if* $\varphi v \neq []$.

Fact 24 *If M is closed and $\varphi v = [M]$, then* $\widehat{\overline{\varphi + \psi}}(v[\sigma]) = M(\overline{\psi v}\ \widehat{\overline{\varphi + \psi}}(\sigma))$.

Lemma 25 $v_1[\sigma]$ *and* $v_2[\tau]$ *are environment separable if* $v_1 \neq v_2$.

PROOF Let $v_1 \neq v_2$ and choose $\varphi v_1 = [\lambda 0]$ and $\varphi v_2 = [(\lambda 0)(\lambda 0)]$. Let ψ be an environment. With Fact 24 we have:

$$\begin{aligned} \widehat{\overline{\varphi + \psi}}(v_1[\sigma]) &= (\lambda 0)\ ((\overline{\psi v_1})[\widehat{\overline{\varphi + \psi}}(\sigma)]) \\ &\neq ((\lambda 0)(\lambda 0))\ ((\overline{\psi v_2})[\widehat{\overline{\varphi + \psi}}(\sigma)]) \\ &= \widehat{\overline{\varphi + \psi}}(v_2[\sigma]) \end{aligned}$$

■

Lemma 26 $v[\sigma]$ *and* s *are environment separable if* s *is not an instantiation.*

PROOF Let s be a term that is not an instantiation. We choose an environment φ as follows:

- If s is an index or an abstraction, choose $\varphi v = [\lambda 0]$.
- If $s = s_1 s_2$ and s_1 is an index or an application, choose $\varphi v = [\lambda 0]$.
- If $s = s_1 s_2$ and s_1 is an abstraction, choose $\varphi v = [(\lambda 0)(\lambda 0)]$.
- If $s = s_1 s_2$ and $s_1 = v'[\tau]$, choose $\varphi v = \varphi v' = [\lambda 0]$.

The claim follows with Fact 24. ■

Fact 27 $\overline{A + [n] + B}\ [f] \neq \overline{A + [n] + B}\ [g]$ *if* $fn \neq gn$.

Lemma 28 $v[\sigma]$ *and* $v[\tau]$ *are environment separable if* σ *and* τ *are environment separable.*

PROOF Let φ be an environment and n be a number such that $\widehat{\overline{\varphi + \psi}}(\sigma)n \neq \widehat{\overline{\varphi + \psi}}(\tau)n$ for every environment ψ. Let φ' be the environment agreeing with φ everywhere except for $\varphi' v = \varphi v + [n]$. We show that φ' separates $v[\sigma]$ and $v[\tau]$. Let ψ be an environment.

$$\begin{aligned} \widehat{\overline{\varphi' + \psi}}\,(v[\sigma]) &= (\widehat{\overline{\varphi' + \psi}}(v))\ [\widehat{\overline{\varphi' + \psi}}(\sigma)] \\ &= \overline{(\varphi' + \psi)(v)}\ [\widehat{\overline{\varphi' + \psi}}(\sigma)] && \text{Fact 23} \\ &= \overline{\varphi v + [n] + \psi v}\ [\widehat{\overline{\varphi' + \psi}}(\sigma)] \\ &\neq \overline{\varphi v + [n] + \psi v}\ [\widehat{\overline{\varphi' + \psi}}(\tau)] && \text{Fact 27 and assumption} \\ &= \widehat{\overline{\varphi' + \psi}}\,(v[\tau]) \end{aligned}$$

■

Theorem 29 (Separation) *There is an algorithm that given an apartness derivation constructs a separating environment.*

PROOF Consider an apartness derivation for $s \# t$. The algorithm recurses on the apartness derivation and constructs an environment separating s and t. Case analysis.

1. If $\delta s = \delta t$ and neither s nor t is an instantiation, the apartness derivation for $s \# t$ is obtained with the rule for applications or abstractions. The claim follows with the inductive hypothesis.
2. If $\delta s \neq \delta t$ and neither s nor t is an instantiation, every environment separates s and t.
3. If $\delta s \neq \delta t$ and either s or t is an instantiation, the claim follows by Lemma 26.
4. If $s = v_1[\sigma]$ and $t = v_2[\tau]$ and $v_1 \neq v_2$, the claim follows by Lemma 25.
5. If $s = v[\sigma]$ and $t = v[\tau]$, the apartness derivation for $s \# t$ is obtained with an apartness derivation for $\sigma@n \# \tau@n$. The inductive hypothesis gives us an environment separating σ and τ. The claim follows by Lemma 28.

Consider an apartness derivation for $\sigma \# \tau$. It is obtained from an apartness derivation for $\sigma@n \# \tau@n$ for some n. By the previous argument we obtain a separating environment for $\sigma@n$ and $\tau@n$. This environment also separates σ and τ. ■

Corollary 30 *The proof system for apartness of normal expressions is sound and complete for the complement of semantic equivalence.*

PROOF Soundness follows with Theorem 29 and Fact 22. For completeness consider two normal expressions that are not semantically equivalent. By Theorem 18 and the soundness of the proof system for axiomatic equivalence (Fact 16) we obtain an apartness derivation for the expressions. ■

Corollary 31 *The proof system for axiomatic equivalence of normal expressions is sound and complete for semantic equivalence.*

PROOF Soundness is asserted by Fact 16. For completeness consider two normal expressions that are semantically equivalent. By the soundness of the apartness system (Corollary 30) and Theorem 18 we know that there is an equivalence derivation for the expressions. ■

9. Informative Decision Algorithm

Theorem 32 (Informative Decision Algorithm) *There is an algorithm that decides axiomatic equivalence of expressions. If two expressions are not axiomatically equivalent, the algorithm yields a separating regular assignment.*

PROOF Given two term expressions s and t, the informative decision algorithm proceeds as follows.

1. Call the algorithm from Theorem 18 on ηs and ηt.

2. If (1) yields a derivation of $\eta s \equiv \eta t$, we know $s \equiv t$ by Facts 9 and 13.
3. If (1) yields a derivation of $\eta s \# \eta t$, we obtain a separating environment φ for ηs and ηt by the algorithm from Theorem 29.
4. Hence $\overline{\varphi}$ is a separating regular assignment for ηs and ηt by Fact 22.
5. Hence $\overline{\varphi}$ is a separating regular assignment for s and t by Lemma 5 and Facts 13 and 7.
6. Hence s and t are not axiomatically equivalent by Lemma 5.

For substitution expressions the algorithm proceeds analogously. ■

Corollary 33 *Semantic and axiomatic equivalence coincide.*

10. Conclusion

In this paper, we take a fresh look at de Bruijn terms and substitution. Our starting point is the de Bruijn algebra, a two-sorted first-order algebra whose objects are de Bruijn terms (i.e., trees) and de Bruijn substitutions (i.e., functions), and whose operations are the operations of Abadi et al.'s σ-calculus [1]. The de Bruijn algebra may serve as a semantic model for the different variants of the σ-calculus. Our main result is the completeness of an axiomatization of the de Bruijn algebra that is deductively equivalent to the rewrite rules of the σ_{SP}-calculus [3]. Thus the terminating and confluent rewriting system of the σ_{SP}-calculus decides equality in the de Bruijn algebra.

Our interest in the de Bruijn algebra stems from the fact that it can serve as the basic building block for a formal metatheory of the λ-calculus. From this perspective, a decision procedure for the de Bruijn algebra is a decision procedure for equational substitution lemmas. As is, there are two decision procedures: The rewriting-based procedure provided by the σ_{SP}-calculus and the translation-based decision procedure developed in this paper.

The Coq library Autosubst [7] provides automation for many-sorted de Bruijn syntax and substitution (many-sorted as for instance in System F). The completeness result of this paper combined with the confluence and termination results for the σ-calculus suggest that in principle every equational substitution lemma for the untyped λ-calculus can be established with Autosubst's rewriting-based automation tactics.

Motivated by the application of Autosubst to rich syntactic systems, we plan to work on de Bruijn algebra with multiple and mutually recursive types of terms. We hope that an elegant formulation of such a system can be found and that the results of this paper carry over to the generalized system.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.

[2] R. Adams. Formalized metatheory with terms represented by an indexed family of types. In *Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2006.

[3] P. Curien, T. Hardin, and J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *J. ACM*, 43(2):362–397, 1996.

[4] P.-L. Curien, T. Hardin, and A. Ríos. Strong normalization of substitutions. In *Mathematical Foundations of Computer Science 1992*, volume 629 of *Lecture Notes in Computer Science*, pages 209–217. Springer Berlin Heidelberg, 1992.

[5] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75 (5):381 – 392, 1972.

[6] G. Dowek, T. Hardin, and C. Kirchner. Higher order unification via explicit substitutions. *Information and Computation*, 157(1–2):183 – 235, 2000.

[7] S. Schäfer and T. Tebbi. Autosubst: Automation for de Bruijn syntax and substitution in Coq. Available at www.ps.uni-saarland.de/autosubst, August 2014.