International Journal of Foundations of Computer Science Vol. 6 No. 4 (1995) 417–430

THE QUINE-BERNAYS COMBINATORY CALCULUS

N. RAJA
Computer Science Group, Tata Institute of Fundamental Research,
Bombay 400 005, India
E-mail: raja@tifrvax.tifr.res.in

and

R. K. SHYAMASUNDAR
Computer Science Group, Tata Institute of Fundamental Research,
Bombay 400 005, India
E-mail: shyam@tifrvax.tifr.res.in

Received 3 November 1994
Revised 4 August 1995
Communicated by R. Parikh

ABSTRACT

We develop a theory for constructing *Combinatory Versions* of λ-calculi. Our theory is based on a method, used by Quine and Bernays, for the general elimination of variables in formulations of first-order logic. Our *Combinatory Calculus* presents a significant departure from those propounded by Schönfinkel and Curry. A non-trivial extension of Quine's technique is developed, to go beyond the realm of first-order quantification theory, and cover the entire λ-calculus. The system consists of five *Combinators*, powerful enough to represent λ-abstractions over arbitrary terms. The *Combinatory Calculus* is shown to have the property of functional completeness. Algorithmic translations from the λ-calculus to the *Combinatory Version*, and vice-versa are provided. The approach has the distinct advantage of being able to encode combinatory formulations of process algebras.

Keywords: Combinatory logic, Lambda-calculus, Functional completeness, Variable-elimination technique.

1. Introduction

The notion of *substitution* repeatedly occurs as a "primitive" in theories of logic and models of programming. On closer examination, substitution appears to be so substantially complex, that one shudders at the thought of attaching the qualifier "primitive" to it anymore. This can be demonstrated easily in the context of λ-calculus.[1] The λ-calculus models applicative behavior using the notion of β-reduction. At the heart of β-reduction lies the mechanism of substitution. Consider $(\lambda x.M)N \xrightarrow{\beta} M[x \leftarrow N]$ where M and N denote terms; $M[x \leftarrow N]$ denotes the result of substituting N for the free occurrences of x in M, and is defined as:

$$x[x \leftarrow N] \equiv N;$$

$$y[x \leftarrow N] \equiv y, \quad \text{if } x \not\equiv y;$$

$$(\lambda y.M_1)[x \leftarrow N] \equiv \lambda y.(M_1[x \leftarrow N]), \quad \text{if } y \not\equiv x \text{ and } y \notin \text{FreeVar}(N);$$

$$(M_1 M_2)[x \leftarrow N] \equiv (M_1[x \leftarrow N])(M_2[x \leftarrow N]).$$

The very complexity of *substitution* has attracted the attention of many researchers who have proposed distinct ways of making it more manageable in different contexts.[2–5] The most remarkable way to completely do away with *substitution* was independently discovered by Schönfinkel[6] and Curry.[7] They showed, in first-order predicate logic, that just two basic combinators — "**S**" and "**K**" — along with appropriately placed parentheses — **")"** and **"("** — are sufficient to do away with the complete apparatus of bound variables, binding constructs, and the substitution mechanism. However, with the introduction of "S" and "K", the expressive power of the resulting combinatory notation goes beyond the confines of first-order logic, and turns into a language for the entire theory of sets and classes. This becomes clear when we find that the very same system can be used to embed the λ-calculus as well.[6] The reason for this 'explosive increase' in expressive power can be traced to the fact that the "S" and "K" combinators support self-application, and further can also operate on each other. Such kind of applicative freedom makes rather extreme semantic demands on the universe in which these combinators reside — requiring such an universe to transcend the boundaries of sets and classes.[6]

Importantly, this technique was designed while keeping in mind that the domain of discourse, (from which the variables have to be eliminated), would satisfy certain properties. For example, it presumes that the terms of the domain (after stripping away the abstraction mechanisms), have a recursively defined <operator><operand> structure, till one reaches the atomic elements. (This structure is evident from the definition of the "S" and "K" combinators.) The λ-calculus is an example of such a structure, where the atomic elements are just variables.

Another ingenious move in completely explaining away *substitution* was made a few decades later independently by Quine[8] and Bernays.[9] They devised new Combinators, which, when introduced in any formulation of first-order logic, could do away with the entire baggage that accompanies variables (bound variables, binding constructs and substitution), while at the same time importantly, without altering the expressive power of the formulation. A significant point of departure of the Combinators designed by Quine and Bernays is that they neither support self application nor do they operate on each other. The combinators act only on the predicates present in the domain of discourse, and give rise to new predicates which are again defined over the original domain alone. The important feature of this technique is that it does not presuppose any kind of term structure of the domain of discourse. However, the technique requires that the domain consist of at least two distinct syntactic sorts; and that the elements of these sorts play distinct roles, say akin to program and data. In fact, in the language of computer science, there is a clear distinction between program and data, and the twain are never confused with each other. Thus, the semantic basis for these combinators can be found within ordinary

set theory. Unfortunately, in spite of their semantic simplicity and tractability, these combinators have remained unnoticed and have thereby not received due attention.

In this paper, we examine the use of the technique of Quine and Bernays to derive a combinatory version of the λ-calculus (a unisorted theory, and the *de facto* programming language used by theorists). At first sight, the goals seem irreconcilable — on the one hand we want to keep the distinction between program and data, while on the other hand the computational power of the λ-calculus is derived very much by blurring this distinction. So, it appears as if we would either land in a system with reduced expressive power, or still worse in an inconsistent one. On the contrary, we demonstrate in this paper that, such a construction, albeit non-trivial, is possible. We build a combinatory version, which while maintaining a clean separation between program and data, uses a strategy of controlled reification to ensure that the requirements of expressive power and consistency are not compromised with. We show that the *Combinatory Calculus* has the property of functional completeness. We provide algorithmic translations from the λ-calculus to the *Combinatory Version*, and vice-versa. We then present a proof of correctness of the two translations up to β-equality.

The organization of the paper is as follows: Sec. 2 briefly reviews some background material; Sec. 3 motivates and introduces a combinatory version, with five basic combinators, for the λ-calculus; Sec. 4 proves functional completeness of the set of combinators, and provides algorithmic translations from the λ-calculus to QBC and vice-versa; Sec. 5 surveys other approaches which eliminate substitution; and finally Sec. 6 concludes the paper.

2. Background

In this section, we review some background material relevant to this work.

2.1. Combinators for a first-order theory

The combinators we design in this paper arise from a technique that was formulated independently by Bernays and Quine. However, there are slight differences in the methods proposed by Bernays and Quine. In this subsection, we give a very brief introduction to the method advocated by Quine.[8,10] This will give the reader a flavor of the *combinators*.

Consider a first-order predicate logic, with:

Alphabet:	$x,\ y,\ z\ldots$	individual variables;
	$a,\ b,\ c\ldots$	individual constants;
	$P,\ Q$	predicate symbols of given arities;
	$\exists$	quantifier.
Terms:	As usual.	
Formulas:	As usual.	

In order to eliminate variables and quantifiers from every formula of the above theory, Quine introduced the *combinators* – *inv, Inv, Ref, Der* – which operate iteratively on the predicates P and Q, to yield new predicates which are in turn defined over the original universe only. The *combinators* are defined as:

$(inv\ P)\ x_1\ \ldots\ x_{n-2}\ x_{n-1}\ x_n$ iff $P\ x_1\ \ldots\ x_{n-2}\ x_n\ x_{n-1}$;
$(Inv\ P)\ x_1\ \ldots\ x_{n-1}\ x_n$ iff $P\ x_n\ x_1\ \ldots\ x_{n-1}$;
$(Ref\ P)\ x_1\ \ldots\ x_n$ iff $P\ x_1\ \ldots\ x_n\ x_n$; and
$(Der\ P)\ x_1\ \ldots\ x_{n-1}$ iff $\exists x_n\ P\ x_1\ \ldots\ x_{n-1}\ x_n$, where x_n is a new variable.

As an example, consider the formula, $\exists x\ Pxyxz$. In order to rid the quantifier $\exists$, and the bound variable x, from this formula, we have to use the combinators. Transform the formula $\exists x\ Pxyxz$ to its equivalent, $\exists x\ (inv\ P)\ xyzx$. Then transform, $\exists x\ (inv\ P)\ xyzx$ to the equivalent formula, $\exists x\ (Ref\ Inv\ inv\ P)\ yzx$. With the use of '*Der*', we achieve the final step of the transformation as $(Der\ Ref\ Inv\ inv\ P)\ yz$, which has neither a quantifier, nor a bound variable.

2.2. The type-free λ-calculus

Terms: Terms are built from variables ($x, x', \ldots \in var$), with the help of the application and lambda-abstraction operations.

$$t\ ::=\ var \mid tt \mid \lambda x.t$$

Axioms:

$t\ =\ t$ for atomic terms;
$(\lambda x.t)\ =\ \lambda y.(t[x \leftarrow y])$ (α-renaming);
$(\lambda x.t_1)t_2\ =\ t_1[x \leftarrow t_2]$ (β-conversion).

Deduction Rules:

$$\frac{t_1 = t_2}{t_1 t_3 = t_2 t_3} \qquad \frac{t_1 = t_2}{t_3 t_1 = t_3 t_2} \qquad \frac{t_1 = t_2}{\lambda x.t_1 = \lambda x.t_2} \qquad \frac{t_1 = t_2}{t_2 = t_1} \qquad \frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}$$

3. Quine-Bernays Combinators for the λ-calculus

In this section, we develop Quine-Bernays Combinatory Calculus (QBC), for the type-free λ-calculus. In the first subsection, we define the calculus formally. The second subsection clarifies the definition of *combinators* and *transformation rules* through a series of illustrative examples.

3.1. Formal Definition of QBC

Conventions 1 *A term of the Quine-Bernays Combinatory Calculus is called a QBC-term. $\vec{C}$ is a notation for any finite string of Basic Combinators. $\vec{t}$ is a notation for any finite string of QBC-terms. ϵ is a notation for an empty string. $M, N, L, t, t_1, t_2 \ldots$ is a syntactic notation for arbitrary QBC-terms. x, y, z is a syntactic notation for arbitrary variables. $t_1 t_2 \ldots t_n$ stands for $(\ldots(t_1 t_2)\ldots t_n)$ (association to the left). The symbol $\equiv$ denotes syntactic equality.*

The formalism of the Quine-Bernays Combinatory Calculus (QBC) is given below:

Notation 1 (Alphabet) *The alphabet of QBC comprises:*

1. `Variables:` $x_0,\ x_1, \ldots$;
2. `Explicit symbol denoting` λ`-calculus application:` $\circ$;
3. `Basic Combinators:` $Act,\ Ref,\ Inv,\ inv,\ Prj$;
4. `Auxiliary symbols:`), (.

Definition 1 (Terms) *QBC-terms are inductively defined:*

1. *Any variable is a term;*
2. $\circ$ *is a term;*
3. *If* $\vec{C}$ *is a finite string of basic combinators, and* $\vec{t}$ *is a finite string of terms, then* $(\vec{C}\ \vec{t}\ )$ *is a term;*
4. *If* t_1 *and* t_2 *are terms, so is* $(t_1 t_2)$.

Transformation Rules 1 (Axiom Schemes) *The axiom schemes for equality in QBC follow:*

Action $(Act\ \vec{C}\ \vec{t}\ )\ M\ =_c\ (\vec{C}\ \vec{t}\ M)$;

Reflection $(Ref\ \vec{C}\ t_1 \ldots\ t_n)\ =_c\ (\vec{C}\ t_1 \ldots\ t_n\ t_n)$;

Major Inversion $(Inv\ \vec{C}\ t_1 \ldots t_{n-1} t_n)\ =_c\ (\vec{C}\ t_n t_1 \ldots t_{n-1})$;

Minor Inversion $(inv\ \vec{C}\ t_1 \ldots t_{n-2} t_{n-1} t_n)\ =_c\ (\vec{C}\ t_1 \ldots t_{n-2} t_n t_{n-1})$;

Projection $(Prj\ \vec{C}\ t_1 \ldots t_{n-1} t_n)\ =_c\ (\vec{C}\ t_1 \ldots t_{n-1})$;

Composition $(\circ\ t_1\ t_2 \ldots\ t_n)\ =_c\ ((t_1\ t_2) \ldots\ t_n)$;

Reification $(t_1 t_2 \ldots\ t_n)\ =_c\ t_1 t_2 \ldots\ t_n$.

Transformation Rules 2 (Deduction Rules) *The deduction rules follow:*

Reflexivity $M\ =_c\ M$;

Transitivity $M\ =_c\ N,\ N\ =_c\ L\ \Rightarrow\ M\ =_c\ L$.

3.2. Informal Description of the Combinators and Transformation Rules

We introduce the symbol '$\circ$' to explicitly denote the operation of λ-application. This will help us to write λ-terms without the use of parenthesis. Hence, instead of 'xy', '$x(yz)$', and '$(xy)z$', we shall write '$\circ xy$', '$\circ x \circ yz$ ', and '$\circ \circ xyz$' respectively. Though we shall write λ-terms without parenthesis, we shall use parenthesis in the formation of QBC-terms, and represent QBC-application by concatenation.

3.2.1. Combinator "*Act*"

Consider the λ-term $\lambda x. \circ yx$. This represents a term which is ready to accept an input. On being provided with the input "a", it computes to the term $\circ ya$. Note that in the above abstraction, the variable y is free, only x is bound.

We introduce the *Basic Combinator "Act"* to represent the action of accepting an input. The *Action* transformation rule reads:

$$\textsc{Action}\ (Act\ \vec{C}\ \vec{t})\ M\ =_c\ (\vec{C}\ \vec{t}\ M)\ ,$$

where $\vec{C}$ denotes any finite string of *Basic Combinators*, $\vec{t}$ denotes any finite string of QBC-terms, and M denotes any QBC-term. We form the QBC term $(Act \circ y)$ which corresponds to the above λ-abstraction. When the term $(Act \circ y)$ is provided with term a, then by the "*Action*" transformation rule we get:

$$(Act \circ y)\ a\ =_c\ (\circ\ y\ a).$$

3.2.2. Combinator "*Ref*"

Consider the abstraction $(\lambda y. \circ\circ xyy)$. In order to get the equivalent QBC-term, we focus on $\circ\circ\ xyy$ with a view to eliminate the bound variable y.

Now, we introduce the next *Basic Combinator "Ref"* and the "*Reflection*" transformation rule.

$$\textsc{Reflection}\ (Ref\ \vec{C}\ t_1 \ldots\ t_n)\ =_c\ (\vec{C}\ t_1 \ldots\ t_n\ t_n)\ .$$

Note that the combinator *Ref* duplicates the last element in the string of terms. We get

$$(Ref\ \circ\circ\ xy)\ =_c\ (\circ\circ xyy).$$

Next to eliminate the variable y from $(Ref \circ\circ\ xy)$ we use the Combinator *Act* introduced in the last subsection, and form $(Act\ Ref \circ\circ\ x)$. The QBC-term $(Act\ Ref \circ\circ\ x)$ corresponds to the λ-term $(\lambda y. \circ\circ xyy)$. The application $(Act\ Ref \circ\circ x)\ a$ leads to:

$$(Act\ Ref \circ\circ\ x)\ a\ =_c\ (Ref \circ\circ\ xa) \quad \text{(Action)}$$

$$=_c\ (\circ\circ\ xaa) \quad \text{(Reflection)}.$$

It can be seen that this is exactly the term, we require.

3.2.3. Discussion

Before we go on to consider more complicated terms, there are certain features of QBC-terms that we wish to point out. A QBC-term could be just a variable or it could have the following structure:

(< String of Basic Combinators> <String of Arguments>).

When a QBC-term is not just a variable, it is enclosed within a pair of parenthesis. Either, or both of the strings within, could be null-strings. The first string is constructed from the following elements — *Act, Ref, Inv, inv, Prj* — and it is devoid of parenthesis. It can be considered as forming the *program segment* of the QBC-term under consideration. Every basic combinator that is part of this string, can be thought of as a programming command. During the execution of the program (which corresponds to term reduction following the rewrite transformation rules), no element of the *program segment* acts on any other component of the *program segment*. In the language of computer science, the architecture follows a '*von Neumann Style*', where the program statements are kept distinct from the data part, and the program statements do not operate on each other. The <String of Arguments> forms the *data segment* of the QBC-term.

Let us now look at the transformation of QBC-terms. For each QBC-term, the transformation rule corresponding to the leading basic combinator in its *program segment* is applied. The *Reflection* transformation rule shows that the *Ref* combinator affects only the term under consideration. It has no effect whatsoever on the adjacent QBC-terms. This property shall hold for the combinators — *Ref, Inv, inv, Prj* — i.e., the effect of their transformations are purely local. The combinator *Act* is the only one which affects adjacent terms. As shown by the *Action* transformation rule, the *Act* combinator inputs the right-adjacent QBC-term, and places it at the end of the <String of Arguments> of the current term.

3.2.4. Combinators "*Inv*" and "*inv*"

The *Basic Combinators*, "*Inv*" and "*inv*", follow the "*Major Inversion*" and "*Minor Inversion*" transformation rules respectively. Between themselves, they can permute any element of the argument string to an arbitrary position in the string.

3.2.5. Combinator "*Prj*"

With the combinators introduced so far, we will be able to encode only the λI-calculus, In order to be able to capture the λK-calculus, we need the *Basic Combinator "Prj"* and the "*Projection*" transformation rule.

PROJECTION $(Prj\ \vec{C}\ t_1 \ldots t_{n-1} t_n) =_c (\vec{C}\ t_1 \ldots t_{n-1})$.

The combinator *Prj* discards the last element of the <String of Arguments>.

Consider the abstraction $\lambda y.x$. Thus we get the required QBC-term as $(Act\ Prj\ x)$. Notice that, it would have been very much possible to define the combinator "*Prj*" in such a way that it combines the effect of the combinator "*Act*" also. However for simplicity of exposition, we choose not to do so.

3.2.6. Discussion

As mentioned earlier, the technique of Quine and Bernays that we are generalizing, was originally conceived to work in the setting of first-order logic, where there is no notion of self-application. However, the Quine-Bernays Combinators are not limited to that situation. They are also effective in situations where self-application is present a priori, like in the λ-calculus. These combinators can 'capture' the λ-calculus without limiting its expressive power and without falling into inconsistencies. In order to be able to represent self-application we need one more transformation rule.

3.2.7. *"Reification"* transformation Rule

The "execution" of a QBC- term consists in successively applying the transformation rules corresponding to the elements of the <String of Basic Combinators>. The "*Reification*" Transformation Rule comes into play when the <String of Basic Combinators> in a QBC-term has become a null-string.

REIFICATION $(t_1 t_2 \ldots t_n) =_c t_1 t_2 \ldots t_n$.

Note that there is no combinator corresponding to the "Reification" transformation rule. The "Reification" rule removes the outermost parenthesis that was enclosing the current QBC-term, and, in the process transforms each element of the <String of Arguments> into an independently "executing" program unit.

3.2.8. Discussion

It is worth pointing out at this stage, that each element of the <String of Arguments> within a QBC-term is also a QBC-term. Whenever an element of the argument string comprises basic combinators, or if it is of a non-atomic form, then it always begins with a parenthesis. This feature helps in discriminating between the program part and the data part of a QBC-term. QBC-terms could also be of the form $(t_1\ t_2)$, where t_1 and t_2 are themselves QBC- terms. Thus concatenation of two QBC-terms represents application in QBC. Now each of the two terms comprising an application might have their own "program" and "data" segments. Thus, terms of the Quine–Bernays Combinatory Calculus are akin to a conglomeration of independently evolving units of a concurrent system. However, they do not just evolve independently, they interact with each too. The combinator *Act* helps in interaction, while the other four Combinators — *Ref, inv, Inv, Prj* — help in independent evolution.

4. Functional Completeness and Mutual Translations

In this section, we prove *Functional Completeness* for QBC, and provide *translations* from the λ-calculus to QBC and vice versa.

4.1. Functional Completeness

In classical combinatory logic, it is known that, any λ-expression can be translated to an equivalent combinatory expression. This property is known as *Functional Completeness* of combinators. This result forms the basis for compiling functional languages into combinators, which are then executed using combinatory abstract machines.[11,4] We now show that such a property holds for QBC.

Theorem 1 (Functional Completeness of QBC) *For any QBC-term ϕ, and any variable x, there exists a QBC term ψ, such that $\psi x =_c \phi$, and x does not occur in the term ψ.*

Proof. The quickest way to prove functional completeness, is to find QBC-terms K and S such that

$$\forall x\ \forall y\ Kxy\ =_c\ x \text{ and } \forall x\ \forall y\ \forall z\ Sxyz\ =_c\ xz(yz),$$

where x, y, z are arbitrary QBC-terms. We have:

$$K\ =\ (Act\ Act\ Prj)$$

$$S\ =\ (Act\ Inv\ Inv\ Inv\ Act\ Inv\ Inv\ Act\ Ref\ Inv\ inv\ Inv\ inv\ Inv\ inv\ Inv\ \circ\ \circ\ \circ).$$

□

4.2. Minimality of the Set of Combinators

It is easy to see of the combinators — Act, Ref, Inv, inv, and Prj — that none can be defined solely using the rest. Act is the only combinator which provides a way to accept new arguments; Existing arguments can be discarded using Prj only; Copies of arguments can be done only using Ref; and both Inv and inv are necessary and sufficient to permute an arbitrary element to a desired position. Thus, if any of the combinators are dropped then the property of functional completeness is invalidated. Hence, these combinators form a minimal set of combinators.

4.3. Mutual Translations

Before we provide translations across λ-calculus and QBC, we first define a *derived Combinator "Fus"*, and the corresponding "*Fusion*" transformation rule. This derived Combinator will help in considerably simplifying the translation from λ-calculus to QBC.

FUSION $(Fus\ \vec{C}\ t_1\ t_2\ t_3\ t_4\ t_5\ t_6\ t_7\ )\ =_c\ (\vec{C}\ t_1\ t_2\ t_3\ t_6\ t_5\ t_4\ t_7)$.

It is easy to see that Fus is a *derived combinator* in the sense that, its effect can be achieved by using combinations of combinators Inv and inv.

In order to be able to translate the λ-calculus to QBC, we need a way to simulate variable abstraction over QBC-terms. We provide such an *abstraction mechanism* below.

Transformation Rules 3 (Variable Abstraction over QBC-terms) *If x does not occur in the term ϕ, then $\lambda^* x.\phi$ is $(Act\ Prj\ \phi)$; if $\phi \equiv x$, then $\lambda^* x.x$ is (Act); if ϕ is of the form $(\vec{C}\ \vec{t}\ )$, then $\lambda^* x.(\vec{C}\ \vec{t}\ )$ is $\lambda^* x.\ (Act^{(n)}\ \vec{C})\ \vec{t}$ where $Act^{(n)}$ denotes the concatenation of n instances of the combinator Act, where $n = |\vec{C}|$; In the remaining case ϕ is necessarily of the form $\phi'\phi''$ and by induction we may assume that $\lambda^* x.\phi' = \psi'$ and $\lambda^* x.\phi'' = \psi''$, so $\lambda^* x.\phi$ is $(Act\ Ref\ Fus \circ \circ \psi'\ \psi'' \circ)$.*

The following rules use the abstraction mechanism defined above, to provide a translation form λ-calculus to QBC.

Transformation Rules 4 (From λ-calculus to QBC) *The general rules for translating a QBC-term to a λ-term are:*

1. $(x)_C \ \mapsto\ x$;
2. $(t'\ t'')_C \ \mapsto\ \circ\ (t')_C\ (t'')_C$
3. $(\lambda x.t)_C \ \mapsto\ \lambda^* x.(t)_C$

Lemma 1 *For every QBC-term t, the QBC-term $(\lambda^* x.t)$ does not contain the variable x, and $(\lambda^* x.t)x =_c t$.*

Proof. It is obvious from the definition of λ^* that x does not occur in $(\lambda^* x.t)$. The second part of the lemma can be proved by induction on the QBC-term t. □

Lemma 2 *For every QBC-term t, $(\lambda^* x.t) =_c (\lambda^* x.(\lambda^* x.t))x$.*

Proof. Follows from the previous lemma. □

In classical combinatory logic, for a long period of time, the existing translations from λ-terms to combinators [7] were not suitable for implementation purposes. The transformation of Combinators from theoretical pearls to practical tools occurred following the discovery of efficient translation schemes.[11] In a similar vein, the translation from λ-calculus to QBC given above, is not the most efficient one; for it leads to a much more complicated QBC-term than necessary. Surely, more efficient translation schemes must be possible.

Next, we examine the translation from QBC to λ-calculus. There may not exist a λ-term corresponding to every QBC-term. However, if a corresponding λ-term exists, then it is unique, up to α-renaming. A similar relation holds between Classical Combinatory Logic and λ-calculus.

Transformation Rules 5 (From QBC to λ-calculus) *The general rules for translating a QBC-term to a λ-term are:*

1. $[x]_\lambda \ \mapsto\ x$;
2. $[\circ\ t'\ t'']_\lambda \ \mapsto\ ([t']_\lambda [t'']_\lambda)$;
3. $[(Act\ \vec{C}\ \vec{t})]_\lambda \ \mapsto\ \lambda x.[(\vec{C}\ \vec{t}\ x)]_\lambda$ `(where` x `is a new variable)`;
4. $[(Ref\ \vec{C}\ t_1 \ldots\ t_n)]_\lambda \ \mapsto\ [(\vec{C}\ t_1 \ldots\ t_n\ t_n)]_\lambda$;

5. $[(Inv\ \vec{C}\ t_1 \ldots t_{n-1} t_n)]_\lambda \ \mapsto\ [(\vec{C}\ t_n t_1 \ldots t_{n-1})]_\lambda$;
6. $[(inv\ \vec{C}\ t_1 \ldots t_{n-2} t_{n-1} t_n)]_\lambda \ \mapsto\ [(\vec{C}\ t_1 \ldots t_{n-2} t_n t_{n-1})]_\lambda$;
7. $[(Prj\ \vec{C}\ t_1 \ldots t_{n-1} t_n)]_\lambda \ \mapsto\ [(\vec{C}\ t_1 \ldots t_{n-1})]_\lambda$;
8. $[t't'']_\lambda \ \mapsto\ [t']_\lambda [t'']_\lambda$.

The reason why certain QBC-terms do not have λ-translations is because, QBC allows the symbol "$\circ$" (explicitly denoting λ-application) to be a valid QBC-term. However, "$\circ$" by itself is not a λ-term. While QBC-terms such as $(\circ\ t'\ t'')$ have meaningful translations in the λ-calculus; strings such as $(t''\ \circ\ t')$, which are also valid QBC-terms, do not have a corresponding λ-translation.

Lemma 3 *For any t, u which are QBC-terms, when t_λ and u_λ are both defined, $t =_c u$ implies $t_\lambda =_\beta u_\lambda$.*

Proof. By induction on the QBC-term t. □

We now present a proof of correctness of the two translations between λ-terms and QBC-terms up to β-equality.

Theorem 2 (From λ-Calculus to QBC and back) *For every λ-term t, $t_{C\lambda} =_\beta t$.*

Proof. The proof is by induction on t. It is obvious in case t is a variable or $t = uv$. Suppose that $t = \lambda x.u$; then $t_C = \lambda^* x.u_C$. Hence $(t_C)x = u_C$ (Lemma 1). Thus, by Lemma 3, we have $(t_{C\lambda})x =_\beta u_{C\lambda}$, and by the induction hypothesis, $u_{C\lambda} =_\beta u$. It follows that $(t_{C\lambda})x =_\beta u$, and hence $\lambda x.(t_{C\lambda})x =_\beta \lambda x.u = t$. Now $t_C = \lambda^* x.u_C$, hence $\lambda^* x.t_C x =_C t_C$ (Lemma 2). As x is a variable which does not occur free in t_C, we have by Lemma 3, $\lambda x.(t_{C\lambda})x =_\beta t_{C\lambda}$. It follows finally, that $t_{C\lambda} =_\beta \lambda x.(t_{C\lambda})x =_\beta t$. □

5. A Comparative Evaluation

In this section, we first briefly summarize the proposals for avoiding substitution, and then discuss the advantages and disadvantages of the QBC approach developed in this paper. There have been many distinct proposals to decompose 'substitution' into more simple operations. All such proposals can be classified into two major approaches, based on whether they distribute arguments, or utilize environments.[12]

5.1. Distributing Arguments

Classical Combinatory Logic[7] and *Director Strings as Combinators*[5] belong to this approach. Here, the *data* of a *program* is progressively distributed to those parts of the *program* which require it.[12] For example consider the *program* $\lambda x.PQ$. The body of the *program* consists of the 'operator' P and 'operand' Q. When this *program* is presented with the data a, we get, $(\lambda x.PQ)a \ \rightarrow\ P[x \leftarrow a]Q[x \leftarrow a]$, where the *data* a has to be substituted for the variable x in both the parts ('operator'

and 'operand') of the body of the *program*. The same effect can be achieved with the combinator S. Consider $S\ P\ Q\ a\ \rightarrow\ (Pa)(Qa)$. Note that the combinator S distributes the *data* a to both the parts of the *program* without the use of variables and substitution. The combinator K discards copies of the argument that have been made by S, from those parts of the *program* which do not require it.

A close look at S and K, reveals that their structure closely resembles the term structure of the λ-calculus. Hence, they provide condensed representations of the λ-calculus,[13] and also provide deep insights regarding the mathematical nature of models for λ-calculus.[1]

The work on *Director Strings as Combinators* [5] can be considered as an optimization over the above technique. The combinator "^" distributes the argument to both the <operator> and the <operand>; the combinator "/" distributes it only to the <operator>; the combinator "\" distributes it only to the <operand>; and the combinator "–" discards the argument.

5.2. Using Environments

Categorical Combinators,[4] and *Explicit Substitutions*,[2] developed from De Bruijn indexing,[3] belong to this approach. Here, the notion of an *environment* is used.[12] An *environment* is a map from variables to terms. In this approach variables are present, but the operation of substitution is done away with. This is the approach used in the implementation of programming languages. To achieve the effect of the reduction $(\lambda x.P)a\ \rightarrow\ P[x \leftarrow a]$, the term P is *executed* in an *environment* which binds the term a to the variable x. The execution starts with an empty *environment*. This *environment* is distributed from the root of the expression towards its leaves. When a redex is encountered, the 'operand' of the redex is added to the *environment*, and this new *environment* is distributed over the 'operator'.

For example, consider the execution of term $(\lambda x.(\lambda y.AB)C)D$ in the empty environment denoted by [ ].

$$\ll (\lambda x.(\lambda y.AB)C)D \gg [\ ].$$

The execution gives

$$\ll (\lambda y.AB)C \gg [x :\ll D \gg]$$

$$\rightarrow \ll AB \gg [x :\ll D \gg;\ y :\ll C \gg]$$

$$\rightarrow\ (\ll A \gg [x :\ll D \gg, y :\ll C \gg])(\ll B \gg [x :\ll D \gg, y :\ll C \gg]).$$

Thus, the whole environment is distributed as one single unit. The reduction of nested redexes proceeds in parallel, but the arguments are distributed throughout the body of the program, and not just to the places where they are required.

5.3. QBC Approach: Advantages and Disadvantages

The QBC approach belongs to the category of distributing arguments. The important point to be noted about Classical Combinatory Logic, and Director Strings,

is that they can be used only in such calculi, whose terms have a recursively defined <operator><operand> structure. However, in such settings, QBC does not compare favorably with Classical Combinatory Logic. For example, if QBC is used to provide an implementation of λ-calculus, then QBC has the advantage that the operations of Inv, and inv, can be implemented by relocating pointers. On the other hand, the translation defined using QBC is inefficient compared to the known translations into Classical Combinators.[13] Also, by closely mirroring the term structure of the λ-calculus, Classical Combinators share a deep relationship with models of λ-calculus.[1] QBC seems to lack such a connection with the semantics of λ-calculus.

On the other hand, QBC is not limited by the term structure of the domain. The QBC approach is universal in the sense that it can be very easily extended to eliminate variables and substitution from calculi which have an entirely different term structure. One area where QBC looks very promising is in the setting of calculi for concurrency. For instance, consider the various calculi that have been developed to model the behavior of concurrent computing systems. An example of such a calculus is the π-calculus.[14,12] These concurrent calculi are, at least syntactically, very different from the λ-calculus. For the present purposes, it suffices to know that most such calculi do not possess the <operator><operand> kind of applicative structure found in the λ-calculus; and secondly, most such calculi are inherently two sorted, the two sorts are, namely, *processes* and *channels.* Further, such calculi have an infinite number of distinct 'abstractors', and also a rich set of operators. The work reported in Ref. 15 uses the QBC approach to derive a combinatory formulation of the π-calculus with replication. The inherently two-sorted theory of the π-calculus provides a very natural setting for the QBC approach.

6. Conclusions

Inspired by an unexplored technique of Quine and Bernays in logic, we generalized a combinatorial definition of set-comprehension terms, to design a novel combinator calculus for functions and substitution, and related these combinators to the untyped λ-calculus. We designed a system of five combinators which could 'capture' the λ-calculus (a unisorted theory), without limiting its expressive power and without lapsing into inconsistencies. We proved that the combinatory formulation is functionally complete, and we also provided algorithmic translations from the λ-calculus to the Combinatory System, and vice-versa.

Even though the gain of QBC with respect to more well known variable elimination methods in sequential systems is still not completely clear, the results obtained are very interesting and encouraging,[15] and the value of (extensions of) QBC ought to be assessed, like this work starts to do. In spite of their simplicity and tractability, the combinators of Quine and Bernays have remained unnoticed and have thereby not received due attention. However, we strongly believe that the further study of these combinators will lead to fruitful insights into the theory and practice of programming and logic.

Acknowledgments

Our thanks to an anonymous referee for invaluable comments. Thanks go to Ms. Margaret D'Souza for typing and typesetting this paper.

References

1. H. Barendregt, "The Lambda calculus", *Studies in Logic* **103** (North-Holland, Amsterdam, 1981).
2. M. Abadi, L. Cardelli, P. L. Curien and J. J. Levy, "Explicit substitutions", *Proc. 17th ACM Annual Symposium on Principles of Programming Languages*, Jan. 1990, pp. 31–46.
3. N. De Bruijn, "Lambda-calculus notation with nameless dummies, A Tool for Automatic Formula Manipulation", *Indag. Math.* **34** (1972) 381–392.
4. P.-L. Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming* (Pitman, 1986).
5. R. Kennaway and R. Sleep, "Director strings as combinators", *ACM Transactions on Programming Languages and Systems*, **10**, 4 (Oct. 1988) 602–626.
6. M. Schönfinkel, "Über die Bausteine der mathematischen Logik", *Math. Annalen* **92** (1924) 305–316. English trans. with an introduction by W. V. Quine in *From Frege to Gödel*, ed. J. van Heijenoort (Harvard Univ. Press, 1967) pp. 355-366.
7. H. B. Curry and R. Feys, *Combinatory Logic*, Vol. 1 (North Holland, 1958).
8. W. V. Quine, "Eliminating variables without applying functions to functions", *Journal of Symbolic Logic* **24**, 4 (Dec. 1959) 324-325.
9. P. Bernays, "Über eine natürliche Erweiterung des Relationenkalkuls", in *Constructivity in Mathematics*, ed. A. Heyting (North-Holland, Amsterdam, 1959).
10. W. V. Quine, "Variables explained away", *Proc. American Philosophical Society*, April 1960.
11. D. A. Turner, "A new implementation technique for applicative languages", *Software Practice and Experience* **9** (1979) 31–49.
12. R. Milner, "Functions as processes", *Research Report* **1154** INRIA Sophia Antipolis, Feb. 1990.
13. D. A. Turner, "Another algorithm for bracket abstraction", *Journal of Symbolic Logic* **44**, 2 (1979) 267–270.
14. R. Milner, J. Parrow and D. Walker, "A calculus of mobile processes, (Parts I and II)", *Information and Computation* **100** (1992) 1–77.
15. N. Raja and R. K. Shyamasundar, "Combinatory formulations of concurrent languages", *Proc. Asian Computing Science Conference*, (to appear) Dec. 1995.