A General Schema for Constructing One-Point Bases in the Lambda Calculus

Mayer Goldberg

BRICS Report Series RS-01-35

ISSN 0909-0878 September 2001

A General Schema For Constructing One-Point Bases in the Lambda Calculus

Mayer Goldberg (gmayer@cs.bgu.ac.il)
Department of Computer Science
Ben Gurion University, Beer Sheva 84105, Israel

Abstract

In this paper, we present a schema for constructing one-point bases for recursively enumerable sets of lambda terms. The novelty of the approach is that we make no assumptions about the terms for which the one-point basis is constructed: They need not be combinators and they may contain constants and free variables. The significance of the construction is twofold: In the context of the lambda calculus, it characterises one-point bases as ways of "packaging" sets of terms into a single term; And in the context of realistic programming languages, it implies that we can define a single procedure that generates any given recursively enumerable set of procedures, constants and free variables in a given programming language.
Keywords: Lambda calculi; Bases; Constants; Scheme Programming Language.

1 Introduction

Intuitively, a basis is a set of λ-terms that generates, by application, all other λ-terms modulo α, β, and η reductions. For example, the set $\{\mathbf{S}, \mathbf{K}\}$, where $\mathbf{S} \equiv \lambda abc.ac(bc)$ and $\mathbf{K} \equiv \lambda ab.a$, is one of the best known bases for the $\lambda\mathbf{K}$-calculus [1, 3, 4, 8]. Historically, the focus on bases has been on how to generate the set of all combinators [2, 3, 4], although it might also be interesting to consider bases for the λ-calculus extended by constants and free variables, because of the presence of literals in real programming languages.

One-point bases (i.e., bases consisting of a single λ-term) constitute classical material by now, addressed, e.g., in Barendregt's textbook on the λ-calculus [1, Chapter 8, Section 5]. They are, however, still something of a Black Art, practised in one of two scenarios:

- Show that a given λ-term X generates $\mathbf{S}$ and $\mathbf{K}$.
- Define a λ-term X that satisfies some given generations of $\mathbf{S}$ and $\mathbf{K}$ from X (e.g., $X(X(XX)) =_{\beta\eta} \mathbf{K}, X(X(X(XX))) =_{\beta\eta} \mathbf{S}$). (See Example 3.3.iii

in Section 3.)

There is no systematic way of tackling these two scenarios in general. Depending on the specifics of each problem and the constraints they impose, varying degrees of intuition, educated guesswork and knowledge about solving systems of equations are required.

In this paper we present a systematic way by which a one-point basis can be constructed from a recursively enumerable set S of λ-terms. The particular choice of terms in S is not important: They can be combinators, or can contain constants and free variables. We make no assumptions about their type or structure, and consequently, we generate them without ever applying them to other terms. The one-point basis we construct is thus a kind of "syntactic packaging" of the terms of S, and this syntactic packaging is the novelty and generality of our approach.

Since our one-point bases can be defined to generate λ-terms that have constants, they are suitable for implementation in programming languages such as Scheme [6] or LISP [7], where they can be used to generate numbers, strings, symbols, procedures, file ports, etc. In Section 5 we define a Scheme procedure `make-one-point-generator` implementing our construction. The procedure packages its arguments into the corresponding one-point basis. For example, the following interaction with a Scheme system illustrates how we generate a one-point basis for the numeral `0` and the successor procedure:

```
> (define X (make-one-point-generator 0 (lambda (n) (+ n 1))))
```

We can now use `X` to generate `0` and the successor procedure, and to apply one to the other:

```
> (X (X X))      ; generating 0
0
> (X ((X X) X)) ; generating the successor procedure
#<procedure>
> ((X ((X X) X))
   (X (X X)))    ; applying the successor procedure to 0
1
```

The details of the Scheme implementation are given in Section 5.

2 Prerequisites and Notation

We assume some familiarity with the untyped λ-calculus [1, 2]. The one-step $\beta\eta$-reduction is denoted $\longrightarrow$, its reflexive and transitive closure is denoted $\longrightarrow\!\!\!\!\!\rightarrow$, and the induced equivalence relation is denoted $=_{\beta\eta}$. The set of all λ-terms is denoted by Λ and the set of all closed λ-terms, i.e., λ-terms with no free variables, also known as *combinators*, is denoted by Λ^0. The Boolean values *true* and *false* are given by

$$\begin{array}{rcl} \mathbf{True} & \equiv & \lambda xy.x \\ \mathbf{False} & \equiv & \lambda xy.y \end{array}$$

respectively. They embed a selection mechanism so that

$$\begin{array}{rcl} (\mathbf{True}\ \mathit{doIfTrue}\ \mathit{doIfFalse}) & \twoheadrightarrow & \mathit{doIfTrue} \\ (\mathbf{False}\ \mathit{doIfTrue}\ \mathit{doIfFalse}) & \twoheadrightarrow & \mathit{doIfFalse} \end{array}$$

For all positive integers n, c_n denotes the n-th Church numeral. The $\mathbf{Succ}_c$ combinator computes the successor function. The $\mathbf{IsZero?}_c$ combinator computes the zero predicate, and the $\mathbf{IsEqual?}_c$ combinator computes the equality predicate. The ordered pair $[x, y]$ is defined as $\lambda z.(zxy)$.

3 Bases

The set of terms *generated* by some set S is the closure of S under application (see [1, Item 8.1.1 (i), Page 165]):

3.1 Definition: *The set of terms generated by some set.* Let $\mathsf{S} \subseteq \Lambda$ be some set of λ-terms. The set S^+ of terms *generated* by S is the smallest set W that satisfies:

- $\mathsf{S} \subseteq \mathsf{W}$.
- For all $M, N \in \mathsf{W}$, $MN \in \mathsf{W}$.

3.2 Definition: *Basis.* (Barendregt [1, Page 165, Definition 8.1.1 (ii)]) A set B is a *basis* for a set W if for all $M \in \mathsf{W}$ there exists $N \in \mathsf{B}^+$ such that $M =_{\beta\eta} N$.

Note that it follows from this definition of a basis that it is possible for a set B to be a basis for a set W, and generate λ-terms that are not in W.

3.3 Examples:

i. Let $\mathbf{S} \equiv \lambda xyz.xz(yz), \mathbf{K} =_{\beta\eta} \lambda xy.x$. The set $\{\mathbf{S}, \mathbf{K}\}$ is a basis for the set of combinators in the $\lambda\mathbf{K}$-calculus.

ii. Let $\mathbf{I} \equiv \lambda x.x, \mathbf{J} \equiv \lambda xyzt.xy(xtz)$. The set $\{\mathbf{I}, \mathbf{J}\}$ is a basis for the set of combinators in the $\lambda\mathbf{I}$-calculus.

iii. Let $X \equiv \lambda x.x\mathbf{SK}$. The set $\{X\}$ is a one-point basis for the set of combinators in the $\lambda\mathbf{K}$-calculus [5, Page 48]. Note that $X(X(XX)) \twoheadrightarrow \mathbf{K}$ and $X(X(X(XX))) \twoheadrightarrow \mathbf{S}$.

4 Constructing a One-Point Basis

4.1 THEOREM: *Let $\mathsf{S} = \{S_k\}_{k\geq 1}$ be a recursively enumerable set of (not necessarily closed) terms, containing at most finitely many constants and free variables. There exists a singleton $\mathsf{X} = \{X\}$ that generates S.*

Proof: Since S is recursively enumerable, there exists a computable surjection $f : \mathbb{N} \to \mathsf{S}$ such that $f(k) = S_k$. Let F be a λ-term that computes f on Church numerals. (Therefore $Fc_k \twoheadrightarrow S_k$.)

We make use of the following property:[1]

$$[P,a][P,b] \quad \twoheadrightarrow \quad PPba \tag{1}$$

We define

$$\begin{aligned} P &\equiv \lambda pba.(\mathbf{IsZero?}_c\ b\ [p,(\mathbf{Succ}_c\ a)]\ (F\ b)) & (2)\\ M_k &\equiv [P,c_k] & (3)\end{aligned}$$

For all $k \geq 0$, we have

$$\begin{array}{rl|rl} M_kM_0 & \equiv [P,c_k][P,c_0] & M_0M_{k+1} & \equiv [P,c_0][P,c_{k+1}] \\ & \equiv (\lambda x.xPc_k)(\lambda x.xPc_0) & & \twoheadrightarrow (\lambda x.xPc_0)(\lambda x.xPc_{k+1}) \\ & \twoheadrightarrow PPc_0c_k & & \twoheadrightarrow PPc_{k+1}c_0 \\ & \twoheadrightarrow [P,(\mathbf{Succ}_c\ c_k)] & & \twoheadrightarrow Fc_{k+1} \\ & =_{\beta\eta} [P,c_{k+1}] & & \twoheadrightarrow S_{k+1} \\ & =_{\beta\eta} M_{k+1} & & \end{array}$$

We now define $X \equiv M_0$. The set $=\{X\}$ is a one-point basis for S, since for all $k > 0$:

$$\begin{aligned} X(\underbrace{X\cdots X}_{k+1}) &\twoheadrightarrow M_0(\underbrace{M_0\cdots M_0}_{k+1}) \\ &=_{\beta\eta} M_0M_k \\ &=_{\beta\eta} S_k & (4)\end{aligned}$$

■

4.2 EXAMPLE: *Generating a one-point basis for the λ-calculus extended with n constants.* Let $\mathsf{Const} = \{const_j\}_{1\leq j\leq n}$ be a set of constants. A basis

[1] In Exercise 6.8.15 (ii) in Barendregt's textbook [1, Page 149], this property is given as a hint for finding a set $\{X_k\}_{k\in\mathbb{N}}$, given a recursive function $f : \mathbb{N}^2 \to \mathbb{N}$ such that $X_nX_m = X_{f(n,m)}$.

for this extended calculus is given by $\mathsf{S} = \{\mathbf{S}, \mathbf{K}\} \cup \mathsf{Const}$. We define the λ-term F, which enumerates the terms of S, as follows:

$$\begin{aligned} F \quad \equiv \quad & \lambda r.(\mathbf{IsEqual?}_c\ r\ c_1\ \mathbf{S} \\ & \quad (\mathbf{IsEqual?}_c\ r\ c_2\ \mathbf{K} \\ & \quad\ (\mathbf{IsEqual?}_c\ r\ c_3\ const_1 \\ & \qquad \ddots \\ & \qquad\quad (\mathbf{IsEqual?}_c\ r\ c_{n+1}\ const_{n-1}\ const_n)))) \end{aligned}$$

We now proceed to define the one-point basis as in the theorem.

In Example 3.3.iii, we presented a one-point basis for the $\lambda\mathbf{K}$-calculus that uses the **S** and **K** combinators. That basis made use of the specific properties of **S** and **K**. In contrast, the one-point bases generated in the following example merely "dispatch" on **S** and **K** without applying them, and hence make no use of their properties.

4.3 EXAMPLE: *Defining one-point bases for the $\lambda\mathbf{K}$-calculus.* As already mentioned, the set $\{\mathbf{S}, \mathbf{K}\}$ is a basis for the $\lambda\mathbf{K}$-calculus. There are two ways to enumerate the terms in this basis, given by F_1 and F_2:

$$\begin{aligned} F_1 & \equiv \lambda r.(\mathbf{IsEqual?}_c\ r\ c_1\ \mathbf{S}\ \mathbf{K}) \\ F_2 & \equiv \lambda r.(\mathbf{IsEqual?}_c\ r\ c_1\ \mathbf{K}\ \mathbf{S}) \end{aligned}$$

Using F_1 to define the corresponding one-point basis $\{X_1\}$, we have:

$$\begin{aligned} X_1(X_1X_1) & =_{\beta\eta} \mathbf{S} \\ X_1(X_1X_1X_1) & =_{\beta\eta} \mathbf{K} \end{aligned}$$

And using F_2 to define the corresponding one-point basis $\{X_2\}$, we have:

$$\begin{aligned} X_2(X_2X_2) & =_{\beta\eta} \mathbf{K} \\ X_2(X_2X_2X_2) & =_{\beta\eta} \mathbf{S} \end{aligned}$$

5 Constructing a One-Point Basis in Scheme

The functional subset of languages such as Scheme and Common LISP provides a suitable setting for working with a one-point basis in the λ-calculus extended by finitely many constants:

- Both languages are modelled on the untyped λ-calculus, and so we can code λ-expressions directly in them.

- Our particular construction of a one-point basis works under any reduction strategy and calling convention, and in particular under Scheme and Common LISP's applicative order.

The λ-term F from Example 4.2 can be encoded directly in Scheme for any n terms, be they numbers, strings, procedures, etc, and the resulting one-point basis is specific to the given F. The following procedure takes the Scheme equivalent of F and returns the term corresponding one-point basis (we have in-lined the construction of ordered pairs):

```
(define make-one-point-generator-lc
  (lambda (F)
    (let ((P (lambda (p)
               (lambda (b)
                   (lambda (a)
                     (if (= b 0)
                         (lambda (x) ((x p) (+ a 1)))
                         (F b)))))))
      (lambda (x) ((x P) 0)))))
```

The procedure can be used as follows:

```
                                ; Defining the term
(define X                       ;   for our 1-point basis
  (make-one-point-generator-lc
    (lambda (n)                 ; This is the dispatcher:
      (case n
        ((1) append)            ;   1 => append procedure
        ((2) reverse)           ;   2 => reverse procedure
        (else '(1 2 3))))))     ;   3 => the list (1 2 3)
> (X (X X))                     ; evaluating to the append
#<procedure append>             ;   procedure
> (X ((X X) X))                 ; evaluating to the reverse
#<procedure reverse>            ;   procedure
> (X (((X X) X) X))             ; evaluating to the list (1 2 3)
(1 2 3)
> ((X (X X))                    ; (append
   ((X ((X X) X))               ;   (reverse
      (X (((X X) X) X)))        ;     '(1 2 3))
   (X (((X X) X) X)))           ;   '(1 2 3))
(3 2 1 1 2 3)
```

The main advantage in the way we defined `make-one-point-generator-lc` is that it provides a faithful rendition, from the λ-calculus into Scheme, of the various terms used in the construction of a one-point basis in the proof of Theorem 3.1: The term F, which maps numerals to terms; The term P (defined in (2)), and finally $X = M_0$ (defined in (3)). Although `make-one-point-generator-lc` uses Scheme numerals rather than Church numerals, this makes no difference here.

An implementation that is natural to Scheme would use Scheme's *variadic* procedures, i.e., procedures that take an arbitrary number of arguments and bind them to a list. Rather than mapping integers to elements of this list we could simply traverse the list. The following procedure does just that:

```
(define make-one-point-generator
  (lambda terms
    (let* ((terms (cons 'initial terms))
           (M (lambda (m)
                (lambda (b)
                  (lambda (a)
                    (if (eq? b terms)
                        (lambda (x) ((x m) (cdr a)))
                        (car b)))))))
      (lambda (x) ((x M) terms)))))
```

The procedure `make-one-point-generator` is called with the terms in the set rather than with the corresponding term F:

```
(define X (make-one-point-generator append reverse '(1 2 3)))
```

The one-point basis `X` can be used as before.

6 Conclusion and Issues

In this paper, we have presented a systematic construction of a one-point basis from any recursively enumerable set S of terms. The novelty of this approach is that it makes no assumptions about the terms in S (and hence it does not apply these terms either). We have provided a Scheme procedure that takes an arbitrary number of arguments and returns a procedure that generates these arguments. The equivalent expression in the λ-calculus can be written to take the n-th Church numeral c_n, followed by n arbitrary terms $S_1, \ldots, S_n$, and the application reduces to a single term that can be used to generate $\{S_1, \ldots, S_n\}$.

Acknowledgements

Thanks go to Henk Barendregt, Olivier Danvy, Daniel Friedman, Julia Lawall, and Larry Moss for their comments and encouragement in the course of this work. This work was completed while visiting BRICS[2] during September 2001.

[2] Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

References

[1] Hendrik P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, 1984.

[2] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[3] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume I. North-Holland Publishing Company, 1958.

[4] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic*, volume II. North-Holland Publishing Company, 1972.

[5] Mayer Goldberg. *Recursive Application Survival in the λ-Calculus.* PhD thesis, Department of Computer Science, Indiana University, June 1996.

[6] Richard Kelsey, William Clinger, and editors Rees, Jonathan. Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[7] Guy L. Jr. Steele. *Common Lisp: The Language.* Digital Press, First edition, 1984.

[8] Joseph Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* The MIT Press Series in Computer Science. MIT Press, 1977.