Solving Equations in the λ-Calculus using Syntactic Encapsulation

Mayer Goldberg

BRICS Report Series RS-95-49

ISSN 0909-0878 September 1995

Solving Equations in the λ-Calculus using Syntactic Encapsulation

Mayer Goldberg
Computer Science Department
Indiana University[*]
(mayer@cs.indiana.edu)

September 15, 1995

Abstract

Syntactic encapsulation is a relation between an expression and one of its sub-expressions, that constraints how the given sub-expression can be used throughout the reduction of the expression. In this paper, we present a class of systems of equations, in which the right-hand side of each equation is syntactically encapsulated in the left-hand side. This class is general enough to allow equations to contain self-application, and to allow unknowns to appear on both sides of the equation. Yet such a system is simple enough to be solvable, and for a solution (though of course not its normal form) to be obtainable in constant time.

Keywords: λ-calculus, programming calculi.

[*]Bloomington, IN 47405, USA. Address for the 1995-1996 academic year: Computer Science Dept., Aarhus University, Ny Munkegade bldg. 540, DK-8000 Århus C, Denmark. This work is supported by the Danish Research Academy, and was completed while visiting BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation).

1 Introduction

1.1 Syntactic Encapsulation and Systems of Equations

In this paper, we introduce the notion of *syntactic encapsulation*, and explore its relevance to solving systems of equations in the untyped λ-calculus.

The central result of the paper is Theorem 3.1, which lists sufficient conditions for the existence of solutions to a particular system of equations. Solving the system characterised by Theorem 3.1 does not involve searching through a space of possible solutions: the solution is generated in constant time.

We present two applications of our main theorem. In the first, we show that a one-point basis can be constructed for the λ–K calculus extended by finitely many constants. In the second application, we show that the problem of filling in a *magic square*, adapted to the λ-calculus, is solvable.

1.2 Prerequisites and Notation

We assume some familiarity with the λ-calculus [1, 2]. Applications are by default left-associative, and λ-abstractions are Curried. $\vec{x}$ abbreviates $x_1, \ldots, x_n$ [1, Item 2.1.3, Page 22] The set of terms *generated* by a set $\mathcal{S}$ is denoted by $\mathcal{S}^+$. The symbol $\oslash$ (pronounced "arb") denotes an arbitrary λ-term. The set of *free variables* in a λ-term M is given by $\mathbf{FreeVars}(M)$. The Boolean *false* and *true* are given by $\lambda xy.y$ and $\lambda xy.x$, respectively. *Negation* is given by $\mathbf{not} = \lambda x.(x\ \mathbf{F}\ \mathbf{T})$. *Conjunction* is denoted by $\mathbf{and} = (\lambda xy.(x\ (y\ \mathbf{T}\ \mathbf{F})\ \mathbf{F}))$. For any terms M, N, the ordered-pair $[M, N]$ is given by $\lambda x.(x\ M\ N)$, where $x \notin \mathbf{FreeVars}(M) \cup \mathbf{FreeVars}(N)$, and the first and second projections are given by $\pi_1^2 = \lambda p.(p\ (\lambda xy.x))$ and $\pi_2^2 = \lambda p.(p\ (\lambda xy.y))$, respectively. A *proper combinator* [5, Chapter 5C] is a term of the form $\lambda\vec{x}.M$, where $M \in \{\vec{x}\}^+$. The n-th Church numeral is denoted by $\ulcorner n \urcorner$. Definitions for $\ulcorner n \urcorner$, the successor, predecessor, test for zero, and test for equality on Church numerals are given by

$$\begin{aligned}
\ulcorner n \urcorner &= \lambda xy.\underbrace{(x \cdots (x}_{n \text{ times}}\ y) \cdots) \\
\mathbf{Succ?}_{\text{Church}} &= \lambda nsz.(s\ (n\ s\ z))
\end{aligned}$$

$$\begin{aligned}
\mathbf{Pred?}_{\text{Church}} &= \lambda n.(\pi^2_2\ (n\ (\lambda p.[(\mathbf{Succ?}_{\text{Church}}\ (\pi^2_1\ p)), (\pi^2_1\ p)])\ [\ulcorner 0 \urcorner, \ulcorner 0 \urcorner])) \\
\mathbf{Zero?}_{\text{Church}} &= \lambda n.(n\ (\mathbf{K}\ \mathbf{F})\ \mathit{True}) \\
\mathbf{Equal?}_{\text{Church}} &= \lambda nm.(\mathbf{and}\ (\mathbf{Zero?}_{\text{Church}}\ (n\ \mathbf{Pred?}_{\text{Church}}\ m)) \\
&\qquad\qquad\qquad (\mathbf{Zero?}_{\text{Church}}\ (m\ \mathbf{Pred?}_{\text{Church}}\ n)))
\end{aligned}$$

respectively. The following combinators are used throughout this paper: $\mathbf{I} = \lambda x.x$, $\mathbf{K} = \lambda xy.x$, $\mathbf{B} = \lambda xyz.(x\ (yz))$, $\mathbf{C} = \lambda xyz.(x\ z\ y)$, $\mathbf{S} = \lambda xyz.(x\ z\ (y\ z))$. Finally, the reflexive, transitive closure of the one-step reduction $\longrightarrow$ is given by $\twoheadrightarrow$.

2 Syntactic Encapsulation

Syntactic encapsulation can be seen as a relation between an expression and one of its sub-expressions, that specifies how the given sub-expression can be used throughout the reduction of the expression:

2.1 **Definition:** *Syntactic Encapsulation.* A λ-term M is said to *syntactically encapsulate* a λ-term N if:

1. N occurs as a subexpression in M.
2. For all M' such that $M \twoheadrightarrow M'$, and N occurs as a sub-expression in M', such an occurrence is not in the functional position of an application.

Occurrences of sub-expressions are, of course, modulo α-equivalence.

When an expression N is syntactically encapsulated in an expression M, no assumption about N is made in M: N can be passed around, returned or discarded, but it can never be applied. In the next section we solve a system of equations, in which expressions on the left-hand side syntactically encapsulate expressions on the right-hand side. When the algorithm for solving such a system of equations is translated into a computer program, the terms on the right-hand side can therefore be of any type: They can be procedures, strings, numbers, arrays, etc. Their type is immaterial, since they will never be used as procedures.

3 Systems of Equations

Many problems in the λ-calculus can be reduced to solving a system of equations of the following form:

$$\begin{cases} (P_1\ x_1 \cdots x_n) & = & Q_1 \\ & \vdots & \\ (P_m\ x_1 \cdots x_n) & = & Q_m \end{cases} \tag{1}$$

where $\mathcal{P} = \{P_j\}_{j=1}^m$ and $\mathcal{Q} = \{Q_j\}_{j=1}^m$ are given for some $m, n \in \mathbb{N}$, and we need to solve for $x_1, \ldots, x_n$. Surprisingly, perhaps, there exist such systems which can be solved without making any assumptions about what $Q_1, \ldots, Q_m$ really are. In such situations, we are able to abstract over the Q_j's, so that we could automatically solve the system for any particular $\{Q_j\}_{j=1}^m$. Such systems *syntactically encapsulate* $Q_1, \ldots, Q_m$.

Of course, not all systems of equations of the form (1) have a solution: For example, when $i \neq j$ and $P_i = P_j$ but not $Q_i = Q_j$, the system is inconsistent, and has no solution. Similarly, if P_i is a sub-expression of P_j, there may or may not be solutions.

The following theorem describes sufficient conditions on $\mathcal{P}$ and $\mathcal{Q}$ for the system in (1) to have a solution.

3.1 **Theorem:** Let $\mathcal{P} = \{P_k = \{\lambda x_1 \cdots x_n.B_k : B_k \in \{x_1, \ldots, x_n\}^+\}\}_{k=1}^m$ be a sequence of m proper combinators (each taking n arguments), and let $\mathcal{Q} = \{Q_k\}_{k=1}^m$ be a sequence of m λ-terms, such that:

- For all $i \neq j$, B_i is not a proper sub-expression of B_j.
- For all i, j, if $P_i = P_j$, then $Q_i = Q_j$.

then the system in (1) can be solved for $\vec{x} = x_1, \ldots, x_n$.

The following facts hold for the given system of equations and its solution:

- If $\mathcal{Q}$ is a sequence of combinators, then $x_1, \ldots, x_n$ can be chosen to be combinators as well.
- Terms in $\mathcal{Q}$ *may* contain as free variables any of $x_1, \ldots, x_n$, for which we are solving.
- For any given system, specified by particular $\mathcal{P}$ and $\mathcal{Q}$, there exist countably many solutions $\vec{x}$ that are not $\alpha\beta\eta$-equivalent to each other.

Before we proceed with the actual proof, we note that since members of $\mathcal{P}$ are *proper combinators* they have the effect of permuting and associating $\vec{x}$ arbitrarily.

Proof: In order to recognise and distinguish between various possible permutations of $x_1, \ldots, x_n$, we need an injection from $\{x_1, \ldots, x_n\}^+$ into a set on which an equality predicate is λ-definable. We choose to use LISP S-expressions [6], so that the solution can translate easily into the Scheme dialect of LISP [3].

We encode the n-th variable using $\ulcorner n \urcorner$, the n-th Church numeral [1, 2], and we encode an application $(M\ N)$ as a pair of the encoding of M and the encoding of N. Since we need to distinguish between variables and applications, we tag encodings of variables with the Boolean $\mathbf{F}$, and encodings of applications with a Boolean $\mathbf{T}$. The j-th variable is thus encoded as $[\mathbf{F}, \ulcorner j \urcorner]$. As a convention, we let $[\mathbf{F}, \ulcorner 0 \urcorner]$ represent the empty list. We now define:

$$\begin{array}{rcl}
\textbf{Empty-List} & = & [\mathbf{F}, \mathbf{F}] \\
\textbf{cons} & = & \lambda ab.[\mathbf{T}, [a, b]] \\
\textbf{car} & = & \lambda x.(\pi^2_1\ (\pi^2_2\ x)) \\
\textbf{cdr} & = & \lambda x.(\pi^2_2\ (\pi^2_2\ x)) \\
\textbf{list} & = & \lambda xy.(\textbf{cons}\ x\ (\textbf{cons}\ y\ \textbf{Empty-List})) \\
\textbf{Encode-Variable} & = & \lambda n.[\mathbf{F}, n] \\
\textbf{atom?} & = & \lambda x.(\textbf{not}\ (\pi^2_1\ x)) \\
\textbf{null?} & = & \lambda x.(\textbf{atom?}\ (\textbf{Zero}?_{\text{Church}}\ (\pi^2_2\ x))\ \mathbf{F}) \\
\textbf{pair?} & = & \lambda x.(\pi^2_1\ x\ \mathbf{T}\ \mathbf{F})\ =\ \pi^2_1 \\
\textbf{equal?} & = & (\Phi\ (\lambda e s_1 s_2.(\textbf{and}\ (\textbf{pair?}\ s_1)\ (\textbf{pair?}\ s_1) \\
 & & \qquad (\textbf{and}\ (e\ (\textbf{car}\ s_1)\ (\textbf{car}\ s_2)) \\
 & & \qquad\qquad (e\ (\textbf{cdr}\ s_1)\ (\textbf{cdr}\ s_2))) \\
 & & \qquad (\textbf{and}\ (\textbf{atom?}\ s_1)\ (\textbf{atom?}\ s_1) \\
 & & \qquad\qquad (\textbf{Equal}?_{\text{Church}}(\pi^2_2\ s_1)\ (\pi^2_2\ s_2)) \\
 & & \qquad\qquad \mathbf{F})))) \\
 & & \text{where } \Phi \text{ is any fixed-point combinator}
\end{array} \tag{2}$$

For each $P_j = (\lambda x_1 \cdots x_n.B_j) \in \mathcal{P}$ we can encode B_j as the list B'_j. For example, let $P_j = \lambda x_1 x_2 x_3.(x_1\ x_3\ (x_2\ x_3))$. We have $B_j = ((x_1\ x_3)\ (x_2\ x_3))$.

The encoding of B_j is given by

$$\begin{aligned} B'_j \;=\; (\mathbf{cons}\ (\mathbf{cons}\ (\mathbf{Encode\text{-}Variable}\ \ulcorner 1 \urcorner) \\ (\mathbf{Encode\text{-}Variable}\ \ulcorner 3 \urcorner)) \\ (\mathbf{cons}\ (\mathbf{Encode\text{-}Variable}\ \ulcorner 2 \urcorner) \\ (\mathbf{Encode\text{-}Variable}\ \ulcorner 3 \urcorner))) \end{aligned} \tag{3}$$

We now construct an environment $\mathbf{env}_{\mathcal{Q}}$ that associates B'_j's with their respective Q_j's. We need to know if a lookup in $\mathbf{env}_{\mathcal{Q}}$ was successful, and so we tag the Q_j's by pairing them with the Boolean value $\mathbf{T}$; If the lookup fails, it doesn't really matter what is returned so long as we can identify the lookup as a failure, so we return $[\mathbf{F}, \oslash]$, which is a pair of the Boolean value $\mathbf{F}$, with *any* λ-term (denoted by $\oslash$). The environment is defined as follows:

$$\begin{aligned} \mathbf{env}_{\mathcal{Q}} \;=\; \lambda x.(\mathbf{equal?}\ x\ B'_1\ [\mathbf{T}, Q_1] \\ (\mathbf{equal?}\ x\ B'_2\ [\mathbf{T}, Q_2] \\ \ddots \\ (\mathbf{equal?}\ x\ B'_m\ [\mathbf{T}, Q_m] \\ [\mathbf{F}, \oslash]))) \end{aligned} \tag{4}$$

As stated earlier, the rôle of the sequence $\mathcal{P}$ is to permute and associate $x_1, \ldots, x_n$. The application $(P_j\ x_1 \ldots x_n)$ needs to construct B'_j so that it could be looked up in the environment $\mathbf{env}_{\mathcal{Q}}$. To accomplish this, we exploit the following property of the standard representation for ordered pairs in the λ-calculus: For any λ-terms M, a, b, we have

$$([M, a]\ [M, b]) \;\twoheadrightarrow\; (M\ M\ b\ a) \tag{5}$$

So M is passed a copy of itself, as well as a and b. We define M as follows:

$$\begin{aligned} M \;=\; \lambda mba.((\lambda c.((\lambda v.(\pi_1^2\ v\ (\pi_2^2\ v) \\ [m, c])) \\ (\mathbf{env}_{\mathcal{Q}}\ c))) \\ (\mathbf{list}\ a\ b)) \end{aligned} \tag{6}$$

Given that a, b in (5) are encodings of either variables or applications, M constructs an encoding of the application $([M, a]\ [M, b])$. This encoding is then looked up in the environment $\mathbf{env}_{\mathcal{Q}}$. If it is found in the environment, then the respective Q_j is returned. Otherwise, M is paired with the new encoding.

If we assume that

$$\{x_1, \ldots, x_n\} \;\cap\; \bigcup_{j=1}^{m} \mathbf{FreeVars}(Q_j) \;=\; \emptyset \tag{7}$$

i.e. that the Q_j's do not contain any of $x_1, \ldots, x_n$ as free variables, we can let

$$x_j \;=\; [M, (\mathbf{Encode\text{-}Variable}\ \ulcorner j \urcorner)] \tag{8}$$

and have

$$\begin{aligned} (P_j\ x_1 \cdots x_n) &\twoheadrightarrow (\mathbf{env}_{\mathcal{Q}}\ B'_j) \\ &\twoheadrightarrow Q_j \end{aligned} \tag{9}$$

However, since the condition in (7) is not a requirement of the theorem, we must *fix* any free occurrence of the $x_1, \ldots, x_n$ in the Q_j's by using mutual fixed-point combinators.

Recall the definition of a mutual fixed-point combinator:

3.2 **Definition:** *Mutual Fixed-Point Combinators.* The λ-terms $\Phi_1, \ldots, \Phi_n$ are said to be *mutual fixed-point combinators* if for any λ-terms $x_1, \ldots, x_n$ we have:

$$\begin{aligned} (\Phi_j\ x_1 \cdots x_n) \;&=\; (x_j\ (\Phi_1\ x_1 \cdots x_n) \cdots (\Phi_n\ x_1 \cdots x_n)) \\ &\quad\ \text{for } j \in \{1, \ldots, n\} \end{aligned} \tag{10}$$

Constructions for mutual fixed-point combinators in the λ-calculus are well-known. For an example, see Barendregt's text [1, Item 6.5.2, Page 142]. Let $\{\Phi_j\}_{j=1}^{n}$ be such a sequence of mutual fixed-point combinators. We now define

$$\begin{aligned} x_j \;=\; &(\Phi_j\ (\lambda x_1 \cdots x_n.[M, (\mathbf{Encode\text{-}Variable}\ \ulcorner 1 \urcorner)]) \\ &\quad \vdots \\ &\quad (\lambda x_1 \cdots x_n.[M, (\mathbf{Encode\text{-}Variable}\ \ulcorner n \urcorner)])) \\ &\text{for } j \in \{1, \ldots, n\} \end{aligned} \tag{11}$$

The solution to the system of equations is given by $\vec{x} = x_1, \ldots, x_n$. Note that each of the x_j syntactically encapsulates $Q_1, \ldots, Q_m$.

Finally, for any particular system of equations given by $\mathcal{P}$ and $\mathcal{Q}$, there exist countably many solutions which are not $\alpha\beta\eta$-equivalent. To show this, we note that no assumption were made throughout the proof, about the value of m, which corresponds to the number of equations in a given system: So long as the conditions on $\mathcal{P}$ and $\mathcal{Q}$ are met, such a system can be solved regardless of m. For a system $\mathcal{S}$ of equations, we can find, using the procedure outlined in the proof, values for $x_1, \ldots, x_n$, for which the system is satisfied. We can extend the system $\mathcal{S}$ into a system $\mathcal{S}'$ by adding additional equations (so long as the constraints on the P_j's and the Q_j's are still met). Using the procedure outlined in the proof, we derive $x'_1, \ldots, x'_n$, which are not $\alpha\beta\eta$-equal to $x_1, \ldots, x_n$, and which solve both $\mathcal{S}'$ as well as $\mathcal{S}$. This completes the proof. ■

3.3 Corollary: A basis of n terms can be reduced into a basis of 1 term, by syntactically encapsulating these n terms.

Proof: Let $\mathcal{Q} = \{Q_j\}_{j=1}^n$ be a sequence of n terms. Let $\mathcal{P} = \{P_j\}_{j=1}^n$ be defined as follows:

$$P_j = \lambda x.(x\ (\underbrace{x \cdots x}_{j+1})) \tag{12}$$

It follows from Theorem 3.1 that there exists a term x such that for $j \in \{1, \ldots, n\}$, we have:

$$(P_j\ x) \twoheadrightarrow (x\ (\underbrace{x \cdots x}_{j+1})) \twoheadrightarrow Q_j \tag{13}$$

So $\{x\}$ is a basis for $\mathcal{Q}$. ■

We now consider two applications of Theorem 3.1 and Corollary 3.3:

3.4 Application: The immediate application of Corollary 3.3 is the construction of a 1-point basis for the λ-calculus with [finitely many] constants [7]. We know that $\{\mathbf{S}, \mathbf{K}\}$ form a basis for the λ-calculus [1, Item 8.1.2, Page 165]. So, for example, let Λ^c be the set of all terms generated by $\mathbf{S}$, $\mathbf{K}$, and a *constant* c. A basis for Λ^c can be generated as follows: Let

$$\left\{ \begin{array}{lcl} Q_1 & = & \mathbf{S} \\ Q_2 & = & \mathbf{K} \\ Q_3 & = & c \end{array} \right. \tag{14}$$

For all $j \in \{1, 2, 3\}$, let P_j be defined as in Corollary 3.3. This corollary guarantees the existence of a term x, such that

$$\left\{ \begin{array}{rcl} (x\ (x\ x)) & \twoheadrightarrow & \mathbf{S} \\ (x\ (x\ x\ x)) & \twoheadrightarrow & \mathbf{K} \\ (x\ (x\ x\ x\ x)) & \twoheadrightarrow & c \end{array} \right. \tag{15}$$

We have the mechanism for encapsulating n terms into a 1-point basis implemented in the Scheme programming language [3]:

```
> (define X (MakeBasis add1 6 "Hello World!"))
> X
#<procedure>
> (X (X X))
#<system procedure 1+>
> (X ((X X) X))
6
> (X (((X X) X) X))
"Hello World!"
> ((X (X X)) (X ((X X) X)))
7
>
```

The above transcript clearly shows why syntactic encapsulation is essential for this application: We must guarantee that constants such as strings and numbers do not appear in the functional position in an application.

3.5 Application: *Tragic Squares.* Consider the problem of filling a *magic square* adapted to the λ-calculus: A *magic square* is an $n \times n$ matrix to be filled with integers. With each magic square we associate a *sum*, which is a number the entries in each row, column and diagonal must add up to. For example, a 3×3 magic square with a sum of 15 can be filled in the following way:

8	1	6
3	5	7
4	9	2

We extend the notion of a magic square to that of a *tragic square*. A tragic square is an $n \times n$ matrix, for which we are given the cover, i.e. a separate value for each row, column and diagonal to add up to. A magic square is simply a special case of a tragic square where all the sums in the cover must be equal, and therefore filling a magic square is a simpler problem than that of filling a tragic square. We can extend the problem to higher dimensions by noting that a $k+1$-dimensional hypercube will have a k-dimensional cover.

Finally, we adapt the problem of filling a tragic square to the λ-calculus: Rather than filling each entry of the $n \times n$ square with an integer, we shall fill each entry with a λ-term; Rather than adding up rows, column and diagonals we shall apply entries to each other, in order, along rows columns and diagonals; And finally, rather than supply a cover of integers to which the rows, columns and diagonals should add up, we shall supply a cover of λ-terms to which the various rows, columns and diagonals should β-reduce. The adapted problem is different in several significant ways from its number-theoretic ancestor:

- Because application is not commutative in the λ-calculus, we need to specify all the values in the cover.

- Because application is not associative in the λ-calculus, we need to specify how the application associates. For example, concerning the first row of the following 3×3 square

x_{11}	x_{12}	x_{13}
x_{21}	x_{22}	x_{23}
x_{31}	x_{32}	x_{33}

  we can apply the entries of $\boxed{x_{11} \mid x_{12} \mid x_{13}}$ in two ways: Either $((x_{11}\ x_{12})\ x_{13})$ or $(x_{11}\ (x_{12}\ x_{13}))$. The order of application of the various rows, columns and diagonals can be handled quite conveniently by choosing the sequence $\mathcal{P} = \{P_k\}_{k=1}$ of proper-combinators appropriately. In fact, by associating a pair $\langle P_k, Q_k \rangle$ with each possible grouping of entries of each row, column and diagonal, it is possible to specify a cover that includes a value for any possible association of entries.

Theorem 3.1 guarantees the existence of countably many solutions for any k-dimensional tragic square, whereas the number-theoretic problem of filling a tragic square doesn't always have a solution.

4 Conclusion and Issues

4.1 Syntactic Encapsulation

In this paper, we introduced the notion of *syntactic encapsulation,* which is essentially a constraint on how a sub-expression can be used throughout the reduction of an expression. Imposing this additional constraint on a system of equations guarantees that the solutions obtained are extremely general.

Theorem 3.1 solves a system of equations by syntactically encapsulating the expressions on the right-hand side of the system. The variables we are solving for, however, may appear as free variables in expressions on the left-hand side as well, which allows for the possibility of circularity in the solutions. In our Ph.D. thesis [4] we explore in detail the implications of such circularity.

4.2 The λ–I Calculus

The λ–I -calculus [2] is a restricted form of the λ-calculus, where the variable of a λ-abstraction must occur free in the body of the λ-abstraction. Thus, for example, $\mathbf{K} = \lambda xy.x$ is not in the λ–I -calculus. Using syntactic encapsulation within the $\lambda - I$ calculus introduces special difficulties, because a general selection mechanism (as in (4)) is not possible (for lack of the $\mathbf{K}$ combinator). When information about the I-solvability [1, Item 2.2.10, Page 41] of the syntactically encapsulated expressions is available, it is often possible to use syntactic encapsulation until a selection becomes necessary, and then use I-solvability, which clearly violates the conditions of syntactic encapsulation. In this manner, however, a one-point basis can be generated, for example, for the λ–I calculus, by syntactically encapsulating the $\mathbf{I}, \mathbf{B}, \mathbf{C}, \mathbf{S}$ combinators, which form a basis for the λ–I -calculus, and all of which are I-solvable.

4.3 One-Point Basis

Several one-point bases are known for the pure λ–I and λ–K calculi. In this paper, however, we show how to construct a one-point basis for a λ-calculus, even if this calculus has been extended with finitely many constants.

We have implemented the mechanism for creating such a one-point basis in the Scheme programming language, and the transcript of Item 3.4 makes it intuitively clear why syntactic encapsulation is needed for this application: Some of the constants we are encapsulating in our example (e.g. strings and integers) cannot be applied to other expressions.

An alternate derivation of a one-point basis, which uses syntactic encapsulation as well, but which can be implemented more efficiently, can be found in our Ph.D. thesis [4].

4.4 Systems of Infinitely-Many Equations

Under certain conditions it is possible to extend Theorem 3.1 to solve systems of infinitely many equations. This is desirable, for example, in order to construct a basis for the λ–K calculus extended by countably many constants. We discuss some results in this area in our Ph.D. thesis [4].

Acknowledgements

I am grateful to BRICS[1] for hosting me during the 1995-1996 academic year, and for providing a stimulating environment. Thanks are also due to Olivier Danvy, Daniel P. Friedman, Julia L. Lawall, and Larry Moss for their comments and encouragement.

References

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

[1]Basic Research in Computer Science, Centre of the Danish National Research Foundation.

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

[3] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[4] Mayer Goldberg. *Recursive Application Survival in the λ-Calculus*. PhD thesis, Department of Computer Science, Indiana University, December 1995. Forthcoming.

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

[6] John McCarthy *et al. LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Massachusetts, 1962.

[7] Joseph Stoy. *Denotational Semantics: the Scott-Strachey approach to programming language theory*. The MIT Press Series in Computer Science. MIT Press, 1977.

Recent Publications in the BRICS Report Series