Lambda-Calculus Schemata*

MICHAEL J. FISCHER[†]   (*fischer-michael@cs.yale.edu*)

Department of Computer Science
Yale University
Box 2158 Yale Station
New Haven, CT 06520–2158, U.S.A.

Keywords: Lambda Calculus, LISP, Continuation-Passing Style, Closures, Functional Objects, Retention Strategy, Deletion Strategy, Bindings, Stack

Abstract. A lambda-calculus schema is an expression of the lambda calculus augmented by uninterpreted constant and operator symbols. It is an abstraction of programming languages such as LISP which permit functions to be passed to and returned from other functions. When given an interpretation for its constant and operator symbols, certain schemata, called *lambda abstractions*, naturally define partial functions over the domain of interpretation. Two implementation strategies are considered: the retention strategy in which all variable bindings are retained until no longer needed (implying the use of some sort of garbage-collected store) and the deletion strategy, modeled after the usual stack implementation of ALGOL 60, in which variable bindings are destroyed when control leaves the procedure (or block) in which they were created. Not all lambda abstractions evaluate correctly under the deletion strategy. Nevertheless, both strategies are equally powerful in the sense that any lambda abstraction can be mechanically translated into another that evaluates correctly under the deletion strategy and defines the same partial function over the domain of interpretation as the original. Proof is by translation into continuation-passing style.

Prologue

The late 1960's and early 1970's were an exciting time for theoretical work in programming languages. Language translation and compilation had been a major focus of the previous decade. Syntax was finally well understood and parsers could be generated automatically from grammars. Semantics was not so well understood, and most real programming languages were defined only by a user's manual and a compiler.

Two general approaches were taken in order to have programming languages with fully specified semantics: (i) Better specification methods were developed that were adequate to fully describe existing large programming languages such as PL/1. (ii) New languages were developed with clean mathematical structure that were more amenable to formal description. McCarthy pioneered the latter approach in basing the LISP 1.5 programming language [21] on a simpler functional language, sometimes called "pure LISP" or M-expressions [20], that was defined in a mathematical style, independent of a particular machine or implementation.

Pure LISP allows the definition and evaluation of functions over S-expressions. The lambda notation for functional abstraction is borrowed from Church's lambda calculus [4], but otherwise there is little similarity between the two systems. Pure LISP has no higher-order functions, and call-by-value evaluation order is implicitly assumed. Two special constructs, conditional expressions and the label operator, allow recursive functions to be defined. Limited as it is, pure LISP is nevertheless powerful enough to express all partial recursive functions and hence provides an adequate basis for a theory of computation [20].

The LISP 1.5 programming language [21] extends pure LISP in many ways that make it more useful in practice but at the same time tend to destroy its clean mathematical properties. Its semantics is defined by an interpreter written in a mixture of pure LISP and English. The distinction between programs and data is blurred. Higher-order functions, assignment, and a global symbol table are added. The simple substitution model used to define pure LISP is replaced by variable bindings based on hierarchical environments. Thus, rather than substituting the actual arguments for the formal parameters when applying a function, the environment is changed by creating associations called *bindings* between the formal parameters and corresponding actual arguments. The environment is consulted whenever the value of a variable is needed.

In the absence of higher-order functions, the environment can be maintained on a simple pushdown stack. New bindings are pushed onto the stack when a function is called, retained during the evaluation of the body of the function, and discarded when done. Functional arguments and values complicate the picture. Free variables in a lambda form should be interpreted in the environment in which the lambda form occurs, not in the environment in effect when the function is eventually applied to some arguments. Because lambda forms are intended to denote functions, any free variables must receive the same interpretation whenever the function is used. A standard method of implementation attaches the caller's environment to the lambda form, resulting in a *closure*. When the closure is later applied to actual arguments, the body of the lambda form is evalu-

ated in the attached environment, after first binding the formal parameters to the actual arguments. Early LISP implementations represent environments explicitly by lists of bindings (called *a*-lists). Whenever the value of a variable is needed, the current *a*-list is searched. Each closure points to its environment *a*-list, and storage occupied by inaccessible environments is recovered by the garbage collector.

MACLISP [22] compromises the semantics of functional arguments in order to achieve greater efficiency by using the "shallow-access" model discussed by Moses [25]. Bindings are stack-based and are discarded when evaluation of the form creating them is complete. Environments are represented by pointers into the stack. This has two consequences for the handling of closures:

- A closure passed as an argument into a function is relatively expensive to evaluate, for the environment must be temporarily changed to the one attached to the closure before evaluating it and restored afterwards, either by temporarily popping the stack to the point of the saved environment or by putting additional bindings on the stack to shadow those that were changed subsequent to the saved environment. Either way, this entails a considerable cost in time and storage.

- A closure cannot be passed out of the function which created it, for the environment of the closure is destroyed when the stack is popped.

Against this backdrop, Hewitt raised the question of just how powerful a shallow-access LISP is [14]. Even pure LISP is universal since its *S*-expressions can easily encode a Turing machine tape, so the question was refined to ask how powerful such a LISP is without using CONS (or any function such as LIST which implicitly uses CONS). In particular, Hewitt asked if it is possible to write a CONS-free function to test if two argument *S*-expressions have the same frontier.

We define the frontier of an *S*-expression to be the list of non-NIL atoms of *S* in sequence. Thus, (A (B C) D), ((A (B C . D))), and (((((A) B) (C D)))) all have the same frontier (A B C D). The frontier is computed by the following LISP function:

```
(DEFUN FRONTIER (X)
  (COND ((NULL X)  NIL)
        ((ATOM X)  (LIST X))
        (T         (APPEND (FRONTIER (CAR X))
                           (FRONTIER (CDR X))))))
```

Here LIST and APPEND have their usual LISP meanings and are easily defined using CONS.

A formal treatment of Hewitt's question requires a way of comparing the expressive power of programming languages. People have the intuitive feeling that some languages are "more powerful" than others. However, most realistic programming languages are universal in the sense that they can simulate Turing machines and hence can compute any partial recursive function. Because of this, attempts to classify real languages according to the functions they can compute inevitably fail, for whatever one language can do, so can another (by simulating the first). This dilemma, in which arguments about relative expressive power become stuck in simulations, and all apparent distinctions between languages evaporate under close scrutiny, is known as the "Turing tar pit" and is a major obstacle towards the development of a comparative theory of programming languages.

Schemata offer a way out of the dilemma. A schema is a program in which primitive data and operations are left unspecified, and two schemata are considered equivalent only if they compute the same result no matter how their constants and basic operators are interpreted. This precludes many of the encoding tricks that lead to the Turing tar pit, and schemata of various kinds can and do differ in expressive power when defined in this way. Program schemata (also called program schemes), were introduced by Ianov [15] and subsequently studied by many others (see [6, 17, 26, 27, 32]). Recursive program schemes, which allow the recursive definition of functions, were also studied [14, 28, 36]. The lambda-calculus schemata of this paper are natural extensions of recursive schemes, obtained through the addition of full lambda abstraction. Good surveys of the state of schematology in 1973 are provided by Chandra [3] and Manna [18].

I became intrigued with Hewitt's question and eventually solved it in the affirmative by reformulating the question as one about lambda-calculus schemata and then showing how to translate any lambda-calculus schema into one that would work correctly under a shallow-access implementation. Since the LISP CONS operator can be represented as a lambda form using Church-encoding of pairs [4], it is possible to translate any LISP program into a lambda-calculus schema which is equivalent to the original program when interpreted over the domain of S-expressions. In the translated program, CONS is only used to construct the S-expression that is the final result of computation. (See Section 6.) If the original LISP program is atomic-valued, then CONS is not used at all.

In the process of solving this problem, I rediscovered the notion of continuation and invented a transformation for converting an arbitrary expression into continuation-passing style (CPS), although I didn't call it that at the time. The first reference to CPS of which I am now aware is a 1966 paper by van Wijngaarden [37]. CPS was also independently discovered by several others over the next several years [19, 23, 24, 30, 35]. Reynolds [31] pro-

vides a fascinating historical account of the early discovery and rediscovery of this concept. His paper alerted me to some of the historical references mentioned above.

This work was presented in preliminary form at the 1972 ACM Conference on Proving Assertions about Programs. I am pleased, after 21 years, to present the final version here.

1.   Introduction

ALGOL 60 can be implemented using a stack for storage of all variables (other than "own"-variables) [8, 29]. Storage is created on the stack when control enters a block and is discarded upon exit. This is sometimes called the "deletion strategy", as the values of the local variables are deleted upon exit from the block as the stack is popped [2]. ALGOL 60 provides no way for these variables subsequently to be referenced, so the deleted variables are no longer needed, and hence, the stack implementation is correct for that language [12]. Other languages such as LISP [21], PAL [9], OREGANO [1], etc., do provide ways in which variables bound in an inner block or procedure may be referenced from outside, so their bindings must be retained; hence the name "retention strategy". Berry has shown that the copy rule of ALGOL, when extended in a natural way to these more powerful languages, is equivalent to the retention strategy and not to the deletion strategy [2].

The retention strategy, then, is seemingly more powerful than the deletion strategy. However, we show in this paper that the classes of programs corresponding to the two strategies are equivalent in a very strong sense— for every retention strategy program, we can find an equivalent program that works correctly under the deletion strategy. Moreover, the translation can be done independently of the particular primitive operations and data which the language happens to contain, so the corresponding "schemata", which are programs in which the primitive operations and data are left unspecified, are equivalent under all interpretations!

To make these ideas more precise, we abstract from the above languages the common feature that they permit procedures which can take other procedures as arguments and return procedures as results. These languages have, in addition, a number of primitive operations defined over some domain of primitive data objects, but we need not concern ourselves with these, for our results will be true for any interpretation of the primitive constant and operator symbols. We thus define lambda-calculus schemata to be lambda-calculus expressions [4], augmented by constant and operator symbols which stand respectively for primitive data elements and operations of the underlying interpretation. Our schemata are similar to the ap-

plicative expressions (AE's) of Landin [16] and the schemata of Hewitt [14] and are defined precisely in the next section.

An interpretation assigns to each constant an element of the domain of interpretation and to each operator a (partial) function on that domain, called an *operation*. Thus, operations are "pure" and do not have side effects. Given an interpretation, closed lambda abstractions, which are schemata of the form $(\lambda x_1 \ldots x_n \,.\, p)$ having no free variables, define partial functions on the domain according to the usual rules of lambda-conversion [4], generalized to multiargument functions. As in LISP, we consider "call-by-value" order of evaluation in which the arguments to a function are evaluated before the function is called. We say that two such schemata are *data-equivalent* if they compute the same partial function on the data domain under all interpretations.

A correct implementation of a lambda-calculus schema requires the retention strategy, for during the course of evaluation, a function may return as a value another function containing free variables. A simple example is the composition functional,

$$\text{COMP} = (\lambda fg \,.\, (\lambda x \,.\, (f \ (g \ x))))$$

which returns the function $(\lambda x \,.\, (f \ (g \ x)))$ containing the free variables f and g. The bindings of f and g established during the application of COMP must be retained as long as the returned function is in existence.

Such a program will not work correctly under the deletion strategy, but any schema which happens never to return as a value another procedure containing free variables will indeed work properly (as will certain others). We call such schemata *deletion-tolerant*. Informally, our main theorem (Theorem 1) states that every closed lambda abstraction is data-equivalent to a deletion-tolerant lambda abstraction which can be effectively obtained from the original lambda abstraction. Thus, although not all programs work correctly in a deletion-strategy implementation, they can be converted to an equivalent form which does work correctly.

2.   Lambda-Calculus Schemata

Lambda-calculus schemata are expressions of the lambda calculus, augmented by uninterpreted names for constants and operations.

Definition 1 *Let* N^+ *be the positive natural numbers. Let* $\mathcal{D}$ *be a set of symbols called* constants. *Let* $\langle \mathcal{F}, \rho \rangle$ *be a ranked alphabet of symbols called* operators, *where* $\rho\colon \mathcal{F} \to \mathsf{N}^+$. *(If* $F \in \mathcal{F}$, *then* $\rho(F)$ *is its* arity, *the number of arguments that it takes.) Let* $\mathcal{X}$ *be a set of symbols called* variables. *We assume that* $\mathcal{D}$, $\mathcal{F}$, *and* $\mathcal{X}$ *are pairwise disjoint.*

A lambda-calculus schema over $\mathcal{D}$, $\mathcal{F}$, and $\mathcal{X}$ *is a member of a formal language whose syntax is given by the BNF grammar shown in Figure 1. The notation* $X \ldots$ *denotes one or more occurrences of* X, *square brackets denote optional items, and curly brackets are used for grouping. We also impose two side conditions that cannot be expressed in BNF:*

1. *In a* ⟨prim-appl⟩ $(F \; q_1 \; \ldots \; q_n)$, n *must equal* $\rho(F)$.

2. *In an* ⟨abstraction⟩ $(\lambda x_1 \; \ldots \; x_n \, . \, p)$, *the variables* $x_1, \ldots, x_n$ *must be distinct.*

⟨schema⟩	::=	⟨variable⟩ | ⟨constant⟩ | ⟨abstraction⟩ | ⟨prim-appl⟩ | ⟨fun-appl⟩ | ⟨conditional⟩
⟨abstraction⟩	::=	'(λ' [⟨variable⟩ ...] '.' ⟨schema⟩ ')'
⟨prim-appl⟩	::=	'(' ⟨operator⟩ ⟨schema⟩ ... ')'
⟨fun-appl⟩	::=	'(' ⟨schema⟩ [⟨schema⟩ ...] ')'
⟨conditional⟩	::=	'(' ⟨schema⟩ '→' ⟨schema⟩ '|' ⟨schema⟩ ')'
⟨variable⟩	::=	x where $x \in \mathcal{X}$
⟨constant⟩	::=	c where $c \in \mathcal{D}$
⟨operator⟩	::=	F where $F \in \mathcal{F}$

Figure 1: Syntax for lambda-calculus schemata.

Expressions are classified according to the syntactic categories to which they belong. Names for the categories are shown in Table 1. We also define $\mathcal{S}$ to be the set of schemata, that is, the set of all expressions belonging to category ⟨schema⟩.

As is usual in the lambda calculus, any variables $x_1, \ldots, x_n$ that occur before the dot in the lambda abstraction $(\lambda x_1 \ldots x_n . p)$ are considered to be formal parameters, and all occurrences of them in the body p are said to be *bound*. A variable is *free* in a schema q if it is not bound by any enclosing lambda abstraction. Let var(p) be the set of variables that appear free in a schema p. We say that p is *closed* if var$(p) = \emptyset$.

We will hereafter always assume that $\mathcal{D}$ contains the two symbols **T** and **F**, representing the truth values 'true' and 'false', respectively, and that $\mathcal{X}$ is countably infinite.

For technical convenience, the above definition does not allow operators to appear without arguments, so in particular, an operator by itself is not a

Table 1: Terminology and notation for syntactic expressions.

Syntactic category	Terminology
⟨schema⟩	*schema*
⟨abstraction⟩	*lambda abstraction*
⟨prim-appl⟩	*primitive application*
⟨fun-appl⟩	*function application*
⟨conditional⟩	*conditional*
⟨variable⟩	*variable*
⟨constant⟩	*constant*
⟨operator⟩	*operator*

schema. However, this does not restrict the power of our schemata since the operator symbol F may be replaced by the equivalent lambda abstraction $(\lambda x_1 \ldots x_n . (F\ x_1\ \ldots\ x_n))$, where $n = \rho(F)$.

A closed lambda-calculus schema becomes a program when given an interpretation for its constants and operators.

Definition 2 *An* interpretation *I is a pair (D, val), where D is the* domain *of the interpretation and* val *is a map which associates to each symbol $c \in \mathcal{D}$ an element $\mathrm{val}(c) \in D$ and to each symbol $F \in \mathcal{F}$ a primitive operation $\mathrm{val}(F)$, which is a partial function in $D^n \to D$, where $n = \rho(F)$. We also require that $\mathrm{val}(\mathbf{T}) \neq \mathrm{val}(\mathbf{F})$.*

The reader will note that this formalism keeps program and data completely separate. The primitive operations associated with primitive operators are defined only for arguments in the data domain. Lambda-calculus schemata, our program elements, are not automatically included in the data domain, nor is it possible to "execute" a data element. Without denying the practical importance of being able to dynamically construct and execute programs, we feel that the distinction between programs and data is useful in analyzing many programming situations, for it allows us to make a systematic static translation of a program while leaving the domain of data on which it operates unchanged. Were programs to reside in the data domain and to be constructed dynamically (as in some dialects of LISP), such a static translation would clearly become impossible.

3. Semantics of Lambda-Calculus Schemata

We define the semantics of lambda-calculus schemata by giving a recursive evaluator, similar to "eval" and "apply" of LISP. Like LISP, we use call-by-value order of evaluation rather than call-by-name, and we do not actually perform string substitution for variables in the schemata but instead maintain an environment which gives the current bindings of the free variables in the schema under consideration. The value of a free variable is obtained when needed from the environment.

3.1. Environments and Closures

A *binding* is an ordered pair (x, v), where $x \in \mathcal{X}$ is a variable and v is the value to which it is bound. An *environment* is a finite set of bindings containing at most one binding for each variable.[1] Thus, we may identify an environment E with a finite function, where $E(x) = v$ if (x, v) is a binding in E, and $E(x)$ is undefined otherwise. We write $\text{var}(E)$ for the set of variables on which E is defined. We use the notation $E[x \mapsto v]$ to denote the environment E' that results from E by binding x to v and leaving bindings for other variables unchanged. Thus, $\text{var}(E') = \text{var}(E) \cup \{x\}$, and for each $y \in \text{var}(E')$,

$$E'(y) = \begin{cases} E(y) & \text{if } y \neq x; \\ v & \text{if } y = x. \end{cases}$$

In case $E = \emptyset$, we may simply write $[x \mapsto v]$ in place of $E[x \mapsto v]$.

The values to which variables can be bound are called *objects*. An object may be either an element from the data domain D or a pair $\langle p, E \rangle$, called a *closure*, where p is a lambda abstraction and E is an environment that contains bindings for all free variables in p.

A closure in many ways behaves like data—it may be the binding of some variable and may be passed to or returned from a function. In addition, it may be applied to some arguments, behaving then like a function. However, closures are of interest to us only as vehicles for defining lambda-schemata evaluation and not as the end products of such evaluations. We consider the ultimate purpose of a program to be the function on the data domain which it defines, and it is this which our transformations on programs are designed to preserve.

Note that environments contain closures and vice versa; hence, their formal definition is by mutual recursion.

---

[1] This is equivalent to the notion of environment used in many LISP implementations in which an environment is a *list* of bindings, with no restrictions on the number of bindings for a given variable, but where the binding actually used is the first one encountered.

Definition 3 *Given an interpretation $I = (D, \text{val})$, we inductively define the set $\mathcal{C}$ of* closures *(on I) and the set $\mathcal{E}$ of* environments *(on I).*

    1. $\emptyset$ (the empty set) $\in \mathcal{E}$.

    2. If $f \in \mathcal{S}$ is a lambda abstraction, $E \in \mathcal{E}$, and each free variable of f is contained in $\text{var}(E)$, then $\langle f, E \rangle \in \mathcal{C}$.

    3. If $E \in \mathcal{E}$, $v \in (D \cup \mathcal{C})$, and $x \in \mathcal{X}$, then $E' = E[x \mapsto v]$ is in $\mathcal{E}$.

For convenience, we let $\mathcal{O}$ denote the set of objects *$(D \cup \mathcal{C})$.*

3.2.   Retention-Strategy Evaluation

Retention-strategy evaluation is defined by a recursive program over a fixed interpretation and defines what we consider to be the "correct" semantics. It takes as arguments a schema and an environment. The result, if the program terminates, is an object in $\mathcal{O}$, called the *value* of the schema. Otherwise, the value is undefined. Our evaluation strategy uses "call-by-value" evaluation order, which means that the actual arguments passed to a function are evaluated before the function is called.

Like LISP, our evaluator is defined by two functions. $\text{ev}_r[p, E]$ evaluates the schema p, interpreting free variables according to the environment E. $\text{ap}_r[c, \boldsymbol{v}]$ applies the closure c to the objects in the vector $\boldsymbol{v}$. Our evaluation rules differ from LISP in that the schemata in the function and argument positions of a function application are evaluated in the same way, and any lambda abstraction that appears without arguments evaluates to a closure, regardless of the position in which it appears.

Definition 4 *Retention-strategy evaluation is specified by two partial functions:*

$$\text{ev}_r : \mathcal{S} \times \mathcal{E} \to \mathcal{O}$$

and

$$\text{ap}_r : \mathcal{O} \times \left(\bigcup_{n \geq 0} \mathcal{O}^n \right) \to \mathcal{O}.$$

Let $p \in \mathcal{S}$, $E \in \mathcal{E}$, $c \in \mathcal{C}$, and $v_1, \ldots, v_n \in \mathcal{O}$. The values of $ev_r[p, E]$ and $ap_r[c, \langle v_1, \ldots, v_n \rangle]$ are defined by the recursive equations shown in Figure 2. These equations are to be interpreted as defining recursive programs for computing ev_r and ap_r.[2]

---

[2] That is, to compute $ev_r[p, E]$, find the first case in the definition whose condition is true for p (if any). Then compute the result specified by that case. In the course of evaluation, it may be necessary to compute ev_r and ap_r for other arguments. These computations are performed recursively. If any of them fail to terminate, or if an infinite chain of recursive calls occurs, or if none of the cases apply to p, or if $p \in \mathcal{X} - \text{var}(E)$, then $ev_r[p, E]$ is undefined. Similar remarks apply to the computation of ap_r.

$$\mathrm{ev}_r[p, E] =_{df} \begin{cases} \mathrm{val}(p) & \text{if } p \in \mathcal{D}; \\ E(p) & \text{if } p \in \mathcal{X}; \\ \langle p, E \rangle & \text{if } p \text{ is a lambda abstraction}; \\ v & \text{if } p = (F\ q_1\ \ldots\ q_n) \text{ is a primitive} \\ & \quad \text{application and } v = \mathrm{val}(F)(v_1, \ldots, v_n), \\ & \quad \text{where } v_i = \mathrm{ev}_r[q_i, E] \in D \text{ for } i = 1, \ldots, n; \\ \xi & \text{if } p = (q_0\ q_1\ \ldots\ q_n) \text{ is a function} \\ & \quad \text{application and } \xi = \mathrm{ap}_r[c, \langle v_1, \ldots, v_n \rangle], \\ & \quad \text{where } c = \mathrm{ev}_r[q_0, E] \in \mathcal{C} \text{ and} \\ & \quad v_i = \mathrm{ev}_r[q_i, E] \in \mathcal{O} \text{ for } i = 1, \ldots, n; \\ \mathrm{ev}_r[q_1, E] & \text{if } p = (b \to q_1 \mid q_2) \text{ and } \mathrm{ev}_r[b, E] = \mathrm{val}(\mathbf{T}); \\ \mathrm{ev}_r[q_2, E] & \text{if } p = (b \to q_1 \mid q_2) \text{ and } \mathrm{ev}_r[b, E] = \mathrm{val}(\mathbf{F}); \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\mathrm{ap}_r[c, \langle v_1, \ldots, v_n \rangle] =_{df} \begin{cases} \mathrm{ev}_r[p, E'] & \text{if } c = \langle (\lambda x_1 \ldots x_n\ .\ p), E \rangle, \text{ where} \\ & \quad E' = E[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Figure 2: Defining equations for retention-strategy evaluation.

We define the partial function on the data domain which is computed by a closed lambda abstraction in a retention-strategy implementation.

Definition 5 *Let $f = (\lambda x_1 \ldots x_n\ .\ p)$ be a lambda abstraction, let $I = (D, \mathrm{val})$ be an interpretation, and let E be an environment such that $\mathrm{var}(E) \supseteq \mathrm{var}(f)$. We associate with the closure $\langle f, E \rangle$ a partial function $\mathrm{fcn}_r(f, E) \colon D^n \to D$ as follows. For each $a_1, \ldots, a_n \in D$, let $b = \mathrm{ap}_r[\langle f, E \rangle, \langle a_1, \ldots, a_n \rangle]$. If $b \in D$, then $\mathrm{fcn}_r(f, E)(a_1, \ldots, a_n) =_{df} b$; otherwise $\mathrm{fcn}_r(f, E)(a_1, \ldots, a_n)$ is undefined.*

If f is closed, then $\mathrm{fcn}_r(f, E)$ does not depend on E, so we omit mention of E and write simply $\mathrm{fcn}_r(f)$.

3.3. Deletion-Strategy Evaluation

A deletion-strategy implementation of lambda-calculus schemata evaluation approximates the correct retention-strategy evaluation. Many programs will work correctly and give the same answers that they would under the retention strategy, while other programs will fail. We can think of a

deletion-strategy implementation as being defined by two functions ev_d and ap_d that only approximate the correct functions ev_r and ap_r. We will neither define precisely a deletion-strategy implementation nor attempt an exact characterization of the set of arguments on which a deletion-strategy implementation is correct. All we need for our purposes is that ev_d and ap_d be correct on a sufficiently broad subclass of programs. In particular, we only require that ev_d (respectively ap_d) be defined and give the correct answer in the case that every value returned by ap_r anywhere in the recursive evaluation of ev_r (respectively ap_r) is an element of the data domain D and never a closure. Thus, we are assuming correctness only when evaluating schemata in which a function is never returned as the value of another function. This restriction, enforced by ALGOL 60, prevents deleted bindings from ever being referenced later in the computation.

Definition 6 *Deletion-strategy evaluation is specified by two recursively defined functions:*

$$\text{ev}_d \colon \mathcal{S} \times \mathcal{E} \to \mathcal{O}$$

and

$$\text{ap}_d \colon \mathcal{O} \times (\bigcup_{n \geq 0} \mathcal{O}^n) \to \mathcal{O}.$$

Let $p \in \mathcal{S}$, $E \in \mathcal{E}$, $c \in \mathcal{C}$, and $v_1, \ldots, v_n \in \mathcal{O}$. The defining equations for ev_d are the same as those shown in Figure 2 for ev_r, except that every occurrence of ev_r and ap_r is replaced by ev_d and ap_d respectively. The defining equation for ap_d is shown in Figure 3.

$$\text{ap}_d[c, \boldsymbol{v}] =_{df} \begin{cases} \text{ev}_d[p, E'] & \text{if } c = \langle (\lambda x_1 \ldots x_n . p), E \rangle \text{ and } \text{ev}_d[p, E'] \\ & \in D, \text{ where } E' = E[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Figure 3: Defining equation for deletion-strategy application.

The partial function computed by a closed lambda abstraction in a deletion-strategy implementation is given by:

Definition 7 *Let $f = (\lambda x_1 \ldots x_n . p)$ be a lambda abstraction, let $I = (D, \text{val})$ be an interpretation, and let E be an environment such that $\text{var}(E) \supseteq \text{var}(f)$. We associate with the closure $\langle f, E \rangle$ the partial function $\text{fcn}_r(f, E) \colon D^n \to D$ defined by*

$$\text{fcn}_d(f, E)(a_1, \ldots, a_n) =_{df} \text{ap}_d[\langle f, E \rangle, \langle a_1, \ldots, a_n \rangle].$$

If f is closed, then $\mathrm{fcn}_d(f, E)$ does not depend on E, so we omit mention of E and write simply $\mathrm{fcn}_d(f)$.

Lemma 1 *Let p be a schema, E an environment, c a closure, and $\boldsymbol{v} \in \mathcal{O}^n$ a vector of objects. Then*

1. *If $\mathrm{ev}_d[p, E]$ is defined, then $\mathrm{ev}_d[p, E] = \mathrm{ev}_r[p, E]$;*

2. *If $\mathrm{ap}_d[c, \boldsymbol{v}]$ is defined, then $\mathrm{ap}_d[c, \boldsymbol{v}] = \mathrm{ap}_r[c, \boldsymbol{v}]$.*

> **Proof (sketch):** Retention- and deletion-strategy evaluation differ only when some closure $\langle f', E' \rangle$ is applied to arguments, and the body of f' evaluates to another closure. In that case, deletion strategy evaluation is undefined. (Cf. Definition 6.) Because of call-by-value evaluation order, this causes the evaluation of the entire schema to be undefined. If this never occurs, then the two evaluation strategies are identical and produce the same result. ∎

We say that a closed lambda abstraction f is *correct* in a deletion-strategy implementation if $\mathrm{fcn}_d(f) = \mathrm{fcn}_r(f)$. It follows easily from the definitions that f is correct in a deletion-strategy implementation if and only if no closure is ever the result of ap_r in the recursive evaluation of $\mathrm{ap}_r[f, \boldsymbol{v}]$, where $\boldsymbol{v}$ is an argument vector at which $\mathrm{fcn}_r(f)$ is defined.

3.4. Safety

We now define a syntactic condition on f that will ensure its correctness in a deletion-strategy implementation.

Definition 8 *A schema p is said to be* safe *if, for every function application $(q_0 \; q_1 \; \ldots \; q_n)$ or primitive application $(F \; q_1 \; \ldots \; q_n)$ that occurs as a subformula of p, each q_i is either a lambda abstraction, a constant, a variable, or a primitive application.*

It follows that every subformula of a safe schema is also safe, so in particular, the body of a safe lambda-abstraction is a safe schema, and a function application or conditional can never appear in the first (function) position of a safe function application, nor can it appear as an argument to a safe function or primitive application. Hence, in a safe schema, the result of a function application or conditional must eventually propagate to the top level to become the final result of the whole evaluation.

We extend the notion of safety to objects and environments in the obvious way.

Definition 9 *The set of* safe objects *and* safe environments *is defined recursively as follows.*

1. $\emptyset$ *(the empty set)* $\in \mathcal{E}$ *is safe.*

2. $v \in D$ *is safe.*

3. $\langle p, E \rangle \in \mathcal{C}$ *is safe if both p and E are safe.*

4. $E' = E[x \mapsto v]$ *in $\mathcal{E}$ is safe if both v and E are safe.*

The following lemma provides a partial converse to Lemma 1 for safe schemata.

Lemma 2 *Let p be a safe schema, E a safe environment, c a safe closure, and $\boldsymbol{v} \in \mathcal{O}^n$ a vector of safe objects. Then*

1. *If $\mathrm{ev}_r[p, E] \in D$ then $\mathrm{ev}_d[p, E] = \mathrm{ev}_r[p, E]$.*

2. *If $\mathrm{ap}_r[c, \boldsymbol{v}] \in D$ then $\mathrm{ap}_d[c, \boldsymbol{v}] = \mathrm{ap}_r[c, \boldsymbol{v}]$.*

   Proof (sketch):   Recall that retention- and deletion-strategy evaluation differ only when the application of some closure to arguments results in another closure. In a safe schema using retention-strategy evaluation, any such result becomes the final result of the whole evaluation. Hence, if the whole expression evaluates to a data element, it must be the case that no application of a closure to arguments results in a closure, in which case retention strategy and deletion strategy evaluation coincide. ∎

Corollary 1 *Let f be a safe closed lambda abstraction. Then $\mathrm{fcn}_d(f) = \mathrm{fcn}_r(f)$.*

3.5.   Equivalence

We are interested in transformations on schemata that preserve meaning. Since we view schemata as vehicles for defining partial functions over the domain of interpretation, we say two schemata are *data equivalent* if they define the same partial function for all interpretations using retention-strategy evaluation.

Definition 10 *We define the relation $\equiv_D$ over pairs of closures, closed lambda abstractions, and data elements, and we call pairs in the relation* data-equivalent.

Let $\langle f, E \rangle$ and $\langle f', E' \rangle$ be closures, where $f = (\lambda x_1 \ldots x_n \, . \, p)$ and $f' = (\lambda x_1 \ldots x_n \, . \, p')$. We write $\langle f, E \rangle \equiv_D \langle f', E' \rangle$ if $\mathrm{fcn}_r(f, E) = \mathrm{fcn}_r(f', E')$

for all interpretations $I = (D, \text{val})$. *That is,* $\text{fcn}_r(f, E)$ *and* $\text{fcn}_r(f', E')$ *are defined at the same points* $(v_1, \ldots, v_n) \in D^n$, *and they agree at those points. Let* f *and* f' *be closed lambda abstractions. We write* $f \equiv_D f'$ *if* $\langle f, \emptyset \rangle \equiv_D \langle f', \emptyset \rangle$. *Let* $a, a' \in D$. *We write* $a \equiv_D a'$ *if* $a = a'$.

The following is immediate from Definitions 5 and 10.

Lemma 3 *Let* f, f' *be closed lambda abstractions. If* $f \equiv_D f'$, *then* $\text{fcn}_r(f) = \text{fcn}_r(f')$.

Data equivalence is a rather coarse relation. For example, the schema

$$p = (\lambda x \,.\, ((> x\ 3) \to (+\ x\ 2) \mid (\lambda x \,.\, x)))$$

is data equivalent to the schema

$$q = (\lambda x \,.\, ((> x\ 3) \to (+\ x\ 2) \mid ((\lambda x \,.\, (x\ x))\ (\lambda x \,.\, (x\ x)))))$$

for both define partial functions whose domain includes only data elements v for which $\text{val}(>)(v, \text{val}(3)) = \text{val}(\mathbf{T})$, and on that domain they both produce the same result $\text{val}(+)(v, \text{val}(2))$. However, if w is a data element for which $\text{val}(>)(w, \text{val}(3)) = \text{val}(\mathbf{F})$, the application of p to w results in a closure, whereas the application of q to w does not terminate.

Data equivalence is difficult to work with because it is not context-independent. That is, suppose p is a closed subformula of a larger formula P, and Q results from P by replacing p with q. Then even though p is data-equivalent to q, it does not follow that P is data-equivalent to Q. For example, let $P = (\lambda x \,.\, ((p\ x)\ x))$ and $Q = (\lambda x \,.\, ((q\ x)\ x))$, where p and q are as in the above example, and let the domain of interpretation be the natural numbers, where $>$, $+$, and numerals receive their usual meanings. Then $p \equiv_D q$, but $P \not\equiv_D Q$, since $\text{fcn}_r(P)(2) = 2$ but $\text{fcn}_r(Q)(2)$ is undefined.

We are thus led to define a stronger notion of equivalence. Intuitively, we say that two schemata are equivalent if there is no context C that can distinguish them in terms of their data behavior. That is, p is equivalent to p' if, for every schema C, then either $(C\ p)$ and $(C\ p')$ both evaluate to the same data element d, or neither evaluates to a data element, and this is true for any environment which binds all free variables in C, p, and p'.

In order to extend our notion of equivalence to closures c and c', we supply the closure to the context C by binding it to a new variable z. This subterfuge is necessary because closures are not a part of the syntax of lambda calculus schemata and the expressions $(C\ c)$ and $(C\ c')$ are not schemata.[3]

---

[3]One could imagine an extension of lambda calculus schemata in which data elements could appear wherever constants are now allowed, and closures could appear wherever

Definition 11 *We define the relation $\equiv$ over pairs of arbitrary schemata, closures, and data elements, and we call pairs in the relation* equivalent.

Let p and p' be schemata (not necessarily closed). We write $p \equiv p'$, if for all interpretations $I = (D, \mathrm{val})$, all schemata C, and all environments E for which $\mathrm{var}(E) \supseteq \mathrm{var}(p) \cup \mathrm{var}(p') \cup \mathrm{var}(C)$, if either $\mathrm{ev}_r[(C\ p), E] \in D$ or $\mathrm{ev}_r[(C\ p'), E] \in D$, then $\mathrm{ev}_r[(C\ p), E] = \mathrm{ev}_r[(C\ p'), E]$.

Let c and c' be closures. We write $c \equiv c'$ if for all interpretations $I = (D, \mathrm{val})$, all schemata C not containing z as a free variable, and all environments E for which $\mathrm{var}(E) \supseteq \mathrm{var}(C)$, if either $\mathrm{ev}_r[(C\ z), E[z \mapsto c]] \in D$ or $\mathrm{ev}_r[(C\ z), E[z \mapsto c']] \in D$, then

$$\mathrm{ev}_r[(C\ z), E[z \mapsto c]] = \mathrm{ev}_r[(C\ z), E[z \mapsto c']].$$

Finally, let $a, a' \in D$. We write $a \equiv a'$ if $a = a'$.

Note that if p and p' are equivalent closed lambda abstractions or equivalent objects, then they are also data equivalent.

4.   Translation to Continuation-Passing Form

We define a function Φ to translate an arbitrary lambda-calculus schema p into a safe schema $\Phi[p]$. The exact semantic relation of p to $\Phi[p]$ is complicated and is the subject of the next section. However, it will be true that if p is a closed schema that evaluates to a data element v, then $\Phi[p]$ evaluates to a closure which, when applied to a closure g, returns the result of applying g to v. Thus, the schemata $(g\ p)$ and $(\Phi[p]\ g)$ will be equivalent for any g. Similar remarks hold if p has free variables, and p is evaluated in an environment in which those variables are bound to data elements.

The intuition behind Φ is to modify the schema so that for any sub-formula g, instead of g passing its evaluation result back to its enclosing context, the context is passed to g as a functional argument called a *continuation*[4]. This requires that g be modified to receive the continuation argument. If g is a lambda abstraction, then an additional parameter is added to g for this purpose. Otherwise, g is turned into a lambda abstraction of one argument. In either case, the modified g applies its continuation argument to the result that g used to return, thereby avoiding the necessity of returning immediately. Syntactically, this avoids the possible safety

_____

primitive operators are now allowed. Such a system would admit a substitution semantics as is commonly used to define the pure lambda calculus and might provide some technical advantages over the operational semantics based on ev_r and ap_r that we use here. We do not pursue this idea further.

[4]The term "continuation" did not appear in the 1972 version of this paper but was coined by others.

violation of having g appear as an argument of another application; hence, the modified schema is safe.

For example, given $(f\ (g\ (h\ a)))$, we could turn it inside-out to get $(\hat{h}\ (\lambda x\ .\ (\hat{g}\ (\lambda y\ .\ (f\ y))\ x))\ a)$, assuming that $\hat{g}$ and $\hat{h}$ are the modified versions of g and h, respectively. Since g and h may be variables which could each be bound to any object whatsoever, it clearly becomes necessary to uniformly add the extra argument to every lambda abstraction appearing in the entire schema, regardless of whether or not it seems to be making the schema unsafe. Thus, we would also replace f by $\hat{f}$ and would add an additional formal parameter k to the call on f which abstracts the context, resulting in $(\lambda k\ .\ (\hat{h}\ (\lambda x\ .\ (\hat{g}\ (\lambda y\ .\ (\hat{f}\ k\ y))\ x))\ a))$.

The actual translation Φ gives a result somewhat different than the above example, for we wish to be able to translate a subformula independently of the context in which it appears. $\Phi[p]$ is roughly equivalent to $\lambda f\,.\,(f\ p))$ and would be exactly equivalent except that, if p happens to return a closure, $\Phi[p]$ applies its argument not to the same closure but to the encoded version which has an additional argument as described above. A precise statement of the relation between p and $\Phi[p]$ is given in Lemma 6.

In the definition of Φ, we also use the auxiliary function Ψ which adds a new argument to a lambda abstraction and translates its body.

Definition 12 Φ *and* Ψ *are transformations on lambda-calculus schemata and are defined recursively. (Note that Φ bears the same relation to Ψ as* ev_r *does to* ap_r.*) The defining equations are shown in Figure 4. The symbols k, g', and $a'_1, \ldots, a'_n$ denote distinct variables that do not appear free in p.*[5]

Some intuition into the translations may be gained by the following examples.

Examples In the following, all symbols $x, a, b, \ldots$ are variables.

   1. $\Phi[x] = (\lambda k\ .\ (k\ x))$.

---

[5]Historical remark: Our choice of placing the continuation parameter first in the modified lambda-abstraction was rather arbitrary; we could just as easily have followed the modern convention of placing it last. The only effect this would have on our transformations is to change $\lambda k x_1 \ldots x_n$ to $\lambda x_1 \ldots x_n k$ in the definition of Ψ, and to change $(g'\ k\ a'_1\ \ldots\ a'_n)$ to $(g'\ a'_1\ \ldots\ a'_n\ k)$ in the definition of Φ. If we were working in a framework in which all multiargument functions were curried, then placing the continuation argument last would permit the definition of $\Psi[(\lambda x_1 \ldots x_n\ .\ p)]$ to be simplified to $(\lambda x_1 \ldots x_n\ .\ \Phi[p])$. But we cannot curry our functions, for converting an application $(g'\ a'_1\ \ldots\ a'_n\ k)$ to its curried form $((\ldots((g'\ a'_1)\ a'_2)\ \ldots\ a'_n)\ k)$ would destroy the safety we are trying to achieve. Further simplifications to our transformation are indeed possible, but they require use of more global information about the expression (see [7, 33]).

$$\Phi[p] =_{df} \begin{cases} (\lambda k \,.\, (k \ p)) & \text{if } p \in \mathcal{D} \cup \mathcal{X}; \\[4pt] (\lambda k \,.\, (k \ \ \Psi[p])) & \text{if } p \text{ is a lambda abstraction}; \\[4pt] (\lambda k \,.\, (\Phi[a_1] \ \ (\lambda a_1'. \\ \qquad\quad \ldots \\ \qquad\quad (\Phi[a_n] \ \ (\lambda a_n' \,.\, (k \ (F \ a_1' \ \ldots \ a_n')))) \ldots))) \\ \qquad\quad \text{if } p = (F \ a_1 \ \ldots \ a_n) \text{ is a primitive} \\ \qquad\quad \text{application}; \\[4pt] (\lambda k \,.\, (\Phi[g] \ \ (\lambda g'. \\ \qquad\quad (\Phi[a_1] \ \ (\lambda a_1'. \\ \qquad\qquad \ldots \\ \qquad\qquad (\Phi[a_n] \ \ (\lambda a_n' \,.\, (g' \ k \ a_1' \ \ldots \ a_n'))) \ldots))))) \\ \qquad\quad \text{if } p = (g \ a_1 \ \ldots \ a_n) \text{ is a function} \\ \qquad\quad \text{application}; \\[4pt] (\lambda k \,.\, (\Phi[a] \ \ (\lambda a' \,.\, (a' \to (\Phi[b] \ \ k) \mid (\Phi[c] \ \ k))))) \\ \qquad\quad \text{if } p = (a \to b \mid c) \text{ is a conditional}; \\[4pt] \text{undefined} & \text{otherwise}. \end{cases}$$

$$\Psi[(\lambda x_1 \ldots x_n \,.\, p)] =_{df} (\lambda k x_1 \ldots x_n \,.\, (\Phi[p] \ \ k)),$$
$$\text{where } k \notin \{x_1, \ldots, x_n\} \text{ and } k \text{ is not free in } p.$$

Figure 4: Defining equations for continuation-passing form transformation.

2. $\Phi[(a \ b)] = (\lambda k \,.\, ((\lambda k \,.\, (k \ a))$
   $\qquad\qquad\qquad (\lambda g' \,.\, ((\lambda k \,.\, (k \ b))$
   $\qquad\qquad\qquad\qquad (\lambda a' \,.\, (g' \ k \ a')))))).$

3. $\Psi[(\lambda x \,.\, a)] = (\lambda k x \,.\, ((\lambda k \,.\, (k \ a))$
   $\qquad\qquad\qquad\quad k)).$

4. $\Phi[(\lambda x \,.\, a)] = (\lambda k \,.\, (k \ \ (\lambda k x \,.\, ((\lambda k \,.\, (k \ a))$
   $\qquad\qquad\qquad\qquad\qquad k)))).$

5. $\Phi[(\lambda x \,.\, (x_1 \ (x_2 \ (x_3 \ x))))] =$
   $\quad (\lambda k \,.\, (k \ (\lambda k x \,.\, ((\lambda k \,.\, ((\lambda k \,.\, (k \ x_1))$
   $\qquad\qquad\qquad\qquad (\lambda g' \,.\, ((\lambda k \,.\, ((\lambda k \,.\, (k \ x_2))$
   $\qquad\qquad\qquad\qquad\qquad (\lambda g' \,.\, ((\lambda k \,.\, ((\lambda k \,.\, (k \ x_3))$
   $\qquad\qquad\qquad\qquad\qquad\qquad (\lambda g' \,.\, ((\lambda k \,.\, (k \ x))$
   $\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda a' \,.\, (g' \ k \ a'))))))$
   $\qquad\qquad\qquad\qquad\qquad (\lambda a' \,.\, (g' \ k \ a'))))))$
   $\qquad\qquad\qquad\qquad (\lambda a' \,.\, (g' \ k \ a'))))))$
   $\qquad\qquad k))))$

5. Equivalence of Retention and Deletion Strategies

In this section we state the properties of the continuation-passing transformation Φ and give our main theorem on the equivalence of power between the two evaluation strategies.

Let p be a closed schema that evaluates to a data element under every interpretation. Then it turns out that $\Phi[p] \equiv_D (\lambda k \, . \, (k \; p))$. However, if p evaluates to a closure c, $\Phi[p]$ is *not* equivalent to $(\lambda k \, . \, (k \; p))$, for $\Phi[p]$ applies its functional argument not to c but to a two-argument "encoding" of c that includes a continuation argument. In that case, $\Phi[p]$ is equivalent to $(\lambda k \, . \, (k \; p^*))$, where p^* is obtained from p by adding an extra continuation argument to every lambda abstraction that occurs in p. For example, if $p = (\lambda x \, . \, (+ \; x \; 3))$, then $\Phi[p]$ is equivalent to $(\lambda k \, . \, (k \; (\lambda h x \, . \, (h \; (+ \; x \; 3)))))$. To state precisely the relation between p and $\Phi[p]$, we need to define the encoding function $()^*$.

5.1. Star Encoding

The encoding function $()^*$ is a bit like the continuation-passing transformation Φ, except that it does not result in a safe schema; it only adds the extra continuation parameter to lambda abstractions and function applications. In greater detail, p^* is obtained from a schema p by replacing each lambda abstraction $(\lambda x_1 \ldots x_n \, . \, q)$ which appears as a subformula of p by $(\lambda k x_1 \ldots x_n \, . \, (k \; q))$, and replacing each function application $(g \; a_1 \; \ldots \; a_n)$ which appears as a subformula of p by $(g \; (\lambda x \, . \, x) \; a_1 \; \ldots \; a_n)$. This transformation is applied recursively to all subformulas, so q, g, and $a_1 \ldots a_n$ are similarly transformed. Thus, we uniformly add a continuation parameter to every lambda abstraction, and we uniformly pass the identity function as the continuation to every (transformed) function application. The effect of this is to make p^* and p evaluate identically, except that if p evaluates to a closure c, then p^* evaluates to a closure c', where $c' \equiv c^*$.[6]

Definition 13 *Let p be a schema. The encoding function p^* is defined recursively by the equation given in Figure 5.*

---

[6]Actually, c' is essentially the same as c^*, differing possibly only in the choice of names for bound variables.

$$p^* =_{df} \begin{cases} p & \text{if } p \in \mathcal{D} \cup \mathcal{X}; \\ (\lambda k x_1 \ldots x_n \, . \, (k \; q^*)) & \\ \qquad \text{if } p = (\lambda x_1 \ldots x_n \, . \, q) \text{ is a lambda abstraction,} \\ \qquad \text{where } k \in \mathcal{X} - (\text{var}(q) \cup \{x_1, \ldots x_n\}); \\ (F \; q_1^* \; \ldots \; q_n^*) & \text{if } p = (F \; q_1 \; \ldots \; q_n) \text{ is a primitive application;} \\ (g^* \; (\lambda x \, . \, x) \; q_1^* \; \ldots \; q_n^*) & \\ \qquad \text{if } p = (g \; q_1 \; \ldots \; q_n) \text{ is a function application;} \\ (a^* \rightarrow b^* \mid c^*) & \text{if } p = (a \rightarrow b \mid c) \text{ is a conditional;} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Figure 5: Defining equation for star encoding.

Definition 14 *We extend* $()^*$ *to objects and environments in the obvious way. Let* $z \in (\mathcal{O} \cup \mathcal{E})$. *Then*

$$z^* =_{df} \begin{cases} z & \text{if } z \in D; \\ \langle f^*, E^* \rangle & \text{if } z = \langle f, E \rangle \in \mathcal{C}; \\ \{(x, v^*) \mid (x, v) \in E\} & \text{if } z = E \in \mathcal{E}; \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The following lemma expresses the intuitively obvious property that ev_r commutes with $()^*$.

Lemma 4 *Let* p *be a schema and* E *an environment such that* $\text{var}(E) \supseteq \text{var}(p)$. *Then*
$$(\text{ev}_r[p, E])^* \equiv \text{ev}_r[p^*, E^*].$$

Proof (sketch): Proof is by induction on the number of steps to compute $\text{ev}_r[p, E]$.

There are three base cases. If $p \in \mathcal{D}$, then $(\text{ev}_r[p, E])^* = (\text{val}(p))^* = \text{val}(p) = \text{val}(p^*) = \text{ev}_r[p^*, E^*]$. If $p \in \mathcal{X}$, then $(\text{ev}_r[p, E])^* = (E(p))^* = E^*(p^*) = \text{ev}_r[p^*, E^*]$. If $p = (\lambda x_1 \ldots x_n \, . \, q)$ is a lambda abstraction, then $(\text{ev}_r[p, E])^* = \langle p, E \rangle^* = \langle p^*, E^* \rangle = \text{ev}_r[p^*, E^*]$. This last step follows since p^* is also a lambda abstraction.

The interesting case is for function application. Let $p = (g \; q_1 \; \ldots \; q_n)$. Suppose that $\text{ev}_r[g, E] = \langle (\lambda x_1 \ldots x_n . r), F \rangle$ for some r and F. (If not, then one can show that $\text{ev}_r[p, E]$ and $\text{ev}_r[p^*, E^*]$ are both undefined.) Let $v_i = \text{ev}_r[q_i, E]$, $i = 1, \ldots, n$. Then

$$\text{ev}_r[p, E] \quad = \quad \text{ev}_r[(g \; q_1 \; \ldots \; q_n), E]$$

$$
\begin{aligned}
&= \quad \mathrm{ap}_r[\langle(\lambda x_1 \ldots x_n . r), F\rangle, \langle v_1, \ldots, v_n\rangle] \\
&= \quad \mathrm{ev}_r[r, F[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]].
\end{aligned} \tag{1}
$$

By definition of star encoding, $p^* = (g^* \ (\lambda x . x) \ q_1^* \ \ldots \ q_n^*)$. By induction, $(\mathrm{ev}_r[g, E])^* \equiv \mathrm{ev}_r[g^*, E^*]$. Hence,

$$
\begin{aligned}
\mathrm{ev}_r[g^*, E^*] &\equiv \quad \langle(\lambda x_1 \ldots x_n . r), F\rangle^* \\
&= \quad \langle(\lambda k x_1 \ldots x_n . (k \ r^*)), F^*\rangle.
\end{aligned} \tag{2}
$$

Let $w_i = \mathrm{ev}_r[q_i^*, E^*]$, and let $\boldsymbol{w} = \langle\langle(\lambda x . x), E^*\rangle, w_1, \ldots, w_n\rangle$. Then applying line 2 gives

$$
\begin{aligned}
\mathrm{ev}_r[p^*, E^*] &= \quad \mathrm{ev}_r[(g^* \ (\lambda x . x) \ q_1^* \ \ldots \ q_n^*), E^*] \\
&\equiv \quad \mathrm{ap}_r[\langle(\lambda k x_1 \ldots x_n . (k \ r^*)), F^*\rangle, \boldsymbol{w}] \\
&= \quad \mathrm{ev}_r[r^*, F^*[x_1 \mapsto w_1] \ldots [x_n \mapsto w_n]].
\end{aligned} \tag{3}
$$

Finally, we have

$$
\begin{aligned}
&(\mathrm{ev}_r[r, F[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]])^* \\
&\equiv \quad \mathrm{ev}_r[r^*, (F[x_1 \mapsto v_1] \ldots [x_n \mapsto v_n])^*] \tag{4} \\
&\equiv \quad \mathrm{ev}_r[r^*, F^*[x_1 \mapsto w_1] \ldots [x_n \mapsto w_n]] \tag{5}
\end{aligned}
$$

Here, line 4 follows by induction, and line 5 uses the facts that $v_i^* \equiv w_i$ for $i = 1, \ldots, n$, which also follow by induction. Combining lines 1, 5, and 3 gives

$$
(\mathrm{ev}_r[p, E])^* = \mathrm{ev}_r[p^*, E^*]
$$

as desired.

The remaining cases are handled similarly. ∎

Corollary 2 *Let p be a schema such that* $\mathrm{var}(p) \subseteq \{x_1, \ldots, x_n\}$. *Then*

$$
(\lambda x_1 \ldots x_n . p^*) \equiv_D (\lambda x_1 \ldots x_n . p).
$$

Proof: The conditions of the corollary ensure that $(\lambda x_1 \ldots x_n . p^*)$ and $(\lambda x_1 \ldots x_n . p)$ are closed lambda-abstractions. Let $v_1, \ldots, v_n \in D$, and let $E = E^* = [x_1 \mapsto v_1] \ldots [x_n \mapsto v_n]$. Let

$$
a = \mathrm{ev}_r[p, E] = \mathrm{ap}_r[\langle(\lambda x_1 \ldots x_n . p), \emptyset\rangle, \langle v_1, \ldots, v_n\rangle]
$$

and

$$
b = \mathrm{ev}_r[p^*, E^*] = \mathrm{ap}_r[\langle(\lambda x_1 \ldots x_n . p^*), \emptyset\rangle, \langle v_1, \ldots, v_n\rangle].
$$

By Lemma 4, $a^* \equiv b$.

If $a \in D$, then $a = a^*$ by Definition 14, and $a^* = b$ follows from Definition 11. Hence, $a = a^* = b$. Conversely, if $b \in D$, then $b = a^* \in$

D. Also, $a^* = a$ by Definition 14 and the fact that $a \in \mathcal{O}$. Again, we have $a = a^* = b$. From Definition 5, it follows that

$$\mathrm{fcn}_r((\lambda x_1 \ldots x_n . p^*)) = \mathrm{fcn}_r((\lambda x_1 \ldots x_n . p)).$$

Hence, Definition 10 gives

$$(\lambda x_1 \ldots x_n . p^*) \equiv_D (\lambda x_1 \ldots x_n . p)$$

as desired. ∎

Lemma 5 *Let* $g, k, a_1, \ldots, a_n$ *be schemata. Then*

$$(g^* \ k \ a_1^* \ \ldots \ a_n^*) \equiv (k \ (g \ a_1 \ \ldots \ a_n)^*).$$

Proof (sketch):   Suppose $g = (\lambda x_1 \ldots x_n . q)$.   Then $g^* = (\lambda k' x_1 \ldots x_n . (k' \ q^*))$, where $k' \notin \mathrm{var}(q^*)$. Hence

$$
\begin{aligned}
(g^* \ k \ a_1^* \ & \ldots \ a_n^*) \\
&= \ ((\lambda k' x_1 \ldots x_n . (k' \ q^*)) \ k \ a_1^* \ \ldots \ a_n^*) \\
&\equiv \ (k \ ((\lambda x_1 \ldots x_n . q^*) \ a_1^* \ \ldots \ a_n^*)) \\
&\equiv \ (k \ ((\lambda k' x_1 \ldots x_n . (k' \ q^*)) \ (\lambda x . x) \ a_1^* \ \ldots \ a_n^*)) \\
&\equiv \ (k \ (g^* \ (\lambda x . x) \ a_1^* \ \ldots \ a_n^*)) \\
&= \ (k \ (g \ a_1 \ \ldots \ a_n)^*)
\end{aligned}
$$

If g is not a lambda abstraction, then by Lemma 4, both g and g^* evaluate to closures or neither do. In the former case, Lemma 4 is used to relate the two, and an argument similar to the one above is applied. In the latter case, both sides of the equivalence are undefined because of our use of call-by-value semantics. ∎

5.2.   Properties of the Continuation-Passing Transformation

The next lemma relates the meaning of Φ to the encoding function $()^*$.

Lemma 6 *Let* p *be a schema, and let* $k \in \mathcal{X} - \mathrm{var}(p)$. *Then*

$$\Phi[p] \equiv (\lambda k . (k \ p^*)).$$

Proof (sketch):   Proof is by induction on the structure of p. The lemma is obvious for $p \in \mathcal{D} \cup \mathcal{X}$.

Suppose p is function application. For simplicity of notation, assume that p has only one argument, so $p = (g \ a)$. By induction, $\Phi[g] \equiv$

$(\lambda k \,.\, (k \; g^*))$ and $\Phi[a] \equiv (\lambda k \,.\, (k \; a^*))$. Then

$$
\begin{aligned}
\Phi[(g \; a)] \quad &= \quad (\lambda k \,.\, (\Phi[g] \;\; (\lambda g' \,.\, (\Phi[a] \;\; (\lambda a' \,.\, (g' \; k \; a')))))) \\
&\equiv \quad (\lambda k \,.\, ((\lambda k \,.\, (k \; g^*)) \;\; (\lambda g' \,.\, (\Phi[a] \;\; (\lambda a' \,.\, (g' \; k \; a')))))) \\
&\equiv \quad (\lambda k \,.\, ((\Phi[a] \;\; (\lambda a' \,.\, (g^* \; k \; a'))))) \\
&\equiv \quad (\lambda k \,.\, ((\lambda k \,.\, (k \; a^*)) \;\; (\lambda a' \,.\, (g^* \; k \; a')))) \\
&\equiv \quad (\lambda k \,.\, (g^* \; k \; a^*)).
\end{aligned}
$$

By Lemma 5, $(g^* \; k \; a^*) \equiv (k \; (g \; a)^*)$. Hence,

$$
\Phi[p] = \Phi[(g \; a)] \equiv (\lambda k \,.\, (k \; (g \; a)^*)) \equiv (\lambda k \,.\, (k \; p^*)).
$$

The other cases are handled similarly. ∎

Lemma 7 *Let p be a schema. Then $\Phi[p]$ is safe.*

> **Proof:** Safety requires that none of the arguments to a function application or primitive application be themselves function applications or conditionals. Inspection of the equations in Figure 4 show that for any schema p, if $\Phi[p]$ is defined, then $\Phi[p]$ is a lambda abstraction. Further inspection then shows that for every function application $q = (q_0 \; q_1 \; \ldots \; q_n)$ or primitive application $(F \; q_1 \; \ldots \; q_n)$ that occurs as a subformula of $\Phi[p]$, each q_i is either a lambda abstraction, a constant, a variable, or a primitive application. Hence, $\Phi[p]$ is safe. ∎

We are now in a position to state and prove our main theorem.

Theorem 1 *Let f be a closed lambda abstraction. We can effectively find a closed lambda abstraction f' such that for all interpretations, $\mathrm{fcn}_d(f') = \mathrm{fcn}_r(f') = \mathrm{fcn}_r(f)$.*

> **Proof:** By assumption, $f = (\lambda x_1 \ldots x_n \,.\, p)$ for some schema p with $\mathrm{var}(p) \subseteq \{x_1, \ldots, x_n\}$. Let $f' = (\lambda x_1 \ldots x_n \,.\, (\Phi[p] \;\; (\lambda x \,.\, x)))$. $\Phi[p]$ is a lambda abstraction, and by Lemma 7, $\Phi[p]$ is safe. Hence, f' is also safe, so $\mathrm{fcn}_d(f') = \mathrm{fcn}_r(f')$ by Corollary 1. From Lemma 6, we have

$$
\begin{aligned}
f' \quad &= \quad (\lambda x_1 \ldots x_n \,.\, (\Phi[p] \;\; (\lambda x \,.\, x))) \\
&\equiv \quad (\lambda x_1 \ldots x_n \,.\, ((\lambda k \,.\, (k \; p^*)) \, (\lambda x \,.\, x)) \\
&\equiv \quad (\lambda x_1 \ldots x_n \,.\, p^*). \qquad\qquad\qquad\qquad (6)
\end{aligned}
$$

By Corollary 2,

$$
f = (\lambda x_1 \ldots x_n \,.\, p) \equiv_D (\lambda x_1 \ldots x_n \,.\, p^*). \qquad\qquad (7)
$$

Lines 6 and 7 give $f \equiv_D f'$, so by Lemma 3, $\mathrm{fcn}_r(f) = \mathrm{fcn}_r(f')$. Hence, $\mathrm{fcn}_d(f') = \mathrm{fcn}_r(f') = \mathrm{fcn}_r(f)$ as desired. ∎

6. Applications to LISP

The original motivation for this work was a question of LISP [20], namely, what can one compute in a deletion-strategy implementation of LISP (such as the "shallow-access" implementation [25]) without using CONS or any function that implicitly uses CONS, but allowing functional arguments (i.e., FUNCTION)?[7] The obvious approach of defining CONS as the Church pairing function (see [4]) does not work directly in a deletion-strategy implementation of LISP. However, it can be made to work by using the CPS transformation described in Section 4 to translate the resulting expression into an equivalent safe form. In this section, we describe in greater detail the relationship between LISP and lambda calculus schemata.

Before proceeding, we need to clarify what we mean by "LISP". Pure LISP, and various dialects such as LISP 1.5 [21], MACLISP [22], and Common LISP [34], all share a common core, but there are significant differences with respect to crucial issues of variable binding strategy and evaluation rules. For example, if F is a variable bound to a function, MACLISP allows the application (F 3), whereas Common LISP requires one to write (FUNCALL F 3).

Rather than tie ourselves to a particular version of LISP, we take an alternate tack. Let S be the set of all LISP S-expressions. Let $\mathcal{D}_S = \{\,'\xi \mid \xi \in S\}$, that is, each member of $\mathcal{D}_S$ consists of an S-expression preceded by a single quote. Let $\mathcal{F}_S = \{\mathtt{cons}, \mathtt{car}, \mathtt{cdr}, \mathtt{eq}, \mathtt{atom}\}$. Let $\mathcal{X}$ be the set of all LISP symbols. Let $I_S = (D_S, \mathrm{val}_S)$ be an interpretation, where $D_S = S$, $\mathrm{val}_S(\mathcal{D}_S)$ maps each constant name of the form $'\xi$ to the S-expression ξ, and $\mathrm{val}_S(\mathcal{F}_S)$ gives each primitive operator its usual LISP interpretation. We identify **T** with $'\mathtt{t}$ and **F** with $'\mathtt{nil}$. We call lambda-calculus schemata over the alphabets $\mathcal{D}$, $\mathcal{F}$, $\mathcal{X}$ and interpreted according to I_S *basic LISP* programs.

For example, the basic LISP program

$$\textit{make-funny-list} =_{df} (\lambda \mathtt{x}\,.\,((\mathtt{atom\,x}) \to (\mathtt{cons\,x\,'foo}) \mid \mathtt{x}))$$

corresponds naturally to the MACLISP program

```
(DEFUN MAKE-FUNNY-LIST (X)
  (COND ((ATOM X) (CONS X 'FOO))
        (T X)))
```

---

[7]CONS is the operation in LISP that allocates a word of storage from the heap. By disallowing CONS, we are prohibiting all explicit use of heap storage by the program, so our question is really about the extent to which memory allocated by FUNCTION can replace heap storage in the context of LISP.

While both seem to be defining a symbol "make-funny-list", there is an important difference. We regard *make-funny-list* as a meta-symbol that abbreviates a particular schema; it is not a part of the language of lambda-calculus schemata. Most versions of LISP do include defined symbols as part of the language and allow them to be used in circular definitions to express recursion, e.g.,

```
(DEFUN LAST (X)
  (COND ((ATOM (CDR X)) (CAR X))
        (T (LAST (CDR X)))))
```

However, recursion can be expressed without circular definitions by using a call-by-value version of the Y-operator (see [10]), so we lose no expressive power by not providing for defined symbols in basic LISP.

We can now reformulate our question as, "What can one compute in a deletion-strategy implementation of basic LISP?" It is clear that CONS *is* required to construct any non-atomic answer that does not happen to already appear in the program as a constant, but if we restrict ourselves to atomic-valued functions, we get:

Theorem 2 CONS-*free basic LISP, even in a deletion-strategy implementation, is universal in the sense that it can compute any computable atomic-valued partial function of S-expressions.*

> **Proof (sketch):** The general method is to first implement the given function in basic LISP. Then we replace all occurrences of cons, car, and cdr by new lambda abstractions *mycons*, *mycar*, and *mycdr*, respectively, which define ordered pairs using Church-encoding [4]. They are defined as follows:
>
> $$mycons =_{df} (\lambda xy \,.\, (\lambda s \,.\, (s \to x \mid y)))$$
> $$mycar \;\; =_{df} (\lambda s \,.\, (s \; \mathbf{T}))$$
> $$mycdr \;\; =_{df} (\lambda s \,.\, (s \; \mathbf{F}))$$
>
> For the modified program to work correctly, it is necessary to Church-encode each input S-expression and each constant. This can be done with the function
>
> $enc =_{df} (\lambda x \,.\, ((\texttt{atom}\, x) \to x \mid (mycons\, (enc\, (\texttt{car}\, x))\, (enc\, (\texttt{cdr}\, x)))))$
>
> Finally, the resulting basic LISP expression is converted by Theorem 1 into a data-equivalent safe program. ■

To obtain a similar theorem for other versions of LISP, one can in principle write a LISP interpreter in basic LISP and then eliminate CONS from it as in the proof of Theorem 2 above.

A consequence of the above theorem is that free storage may be saved at the expense of increased use of the stack. In the absence of `CONS`, the stack must grow arbitrarily large, since some form of unbounded storage is necessary in order to be universal, and the stack is the only storage we are permitting.

Another consequence is that no implementation of LISP which properly handles `FUNARG`'s can use only a "pure" pushdown store; it must have pointers into the stack, use free storage for purposes other than implementing `CONS` (as in the "a-list" implementation), or use some other such device, for we know that a machine with just a pure pushdown store is not universal.

Finally, we obtain as a corollary an automaton-theoretic result (which is also easy to prove directly). A "stack" implementation of LISP may be thought of as a stack automaton in the sense of Ginsburg et al. [11], augmented by the addition of pointers which may point to elements lower down in the stack. (These pointers represent bindings.) A careful analysis of the use of these pointers reveals that the head of the stack automaton need only be able to move around the stack in three ways: down to the next lower element, down to the destination of a given pointer, or up to the top. Theorem 2 says that such a machine is universal, but we know that a stack automaton lacking pointers is not. This contrasts with the fact that pointers do *not* increase the power of a pushdown store machine (see Cole [5] and Hewitt [13]).

Acknowledgement

References

1. Berry, Daniel M. Introduction to Oregano. *SIGPLAN Notices*, 6, 2 (February 1971) 171–190.

2. Berry, Daniel M. Block structure: Retention or deletion. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (1972) 86–100.

3. Chandra, Ashok K. *On the Properties and Applications of Program Schemas.* PhD thesis, Computer Science Department, Stanford University (1973). Also appears as Report No. CS–336, AI–188 (February 1973).

4. Church, Alonzo. *The Calculi of Lambda-Conversion. Annals of Mathematics Studies*, Princeton University Press, Princeton, New Jersey (1941).

5. Cole, Steven N. Pushdown store machines and real-time computation. In *Proceedings of the [First Annual] ACM Symposium on Theory of Computing* (1969) 233–245.

6. Cooper, D. Mathematical proofs about computer programs. In *Machine Intelligence 1*, Oliver and Boyd (1967) 17–28.

7. Danvy, Olivier and Filinski, Andrzej. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2, 4 (December 1992) 361–391.

8. Dijkstra, Edsgar W. Recursive programming. In Rosen, S., editor, *Programming Systems and Languages*, McGraw-Hill, New York (1967).

9. Evans, Arthur. PAL—a language designed for teaching programming linguistics. In *Proceedings of the ACM 23rd National Conference* (August 1968).

10. Friedman, Daniel P. and Felleisen, Matthias. *The Little LISPer.* The MIT Press, trade edition (1988) chapter 9, 156–157.

11. Ginsburg, Seymour, Greibach, Sheila, and Harrison, Michael. One-way stack automata. *Journal of the ACM*, 14, 2 (April 1967) 389–418.

12. Henhapl, W. and Jones, C. D. *The Block Concept and Some Possible Implementations, with Proofs of Equivalence.* Technical Report TR 25.104, IBM Vienna (April 1970).

13. Hewitt, Carl. (1970). Personal communication.

14. Hewitt, Carl. *More Comparative Schematology.* Artificial intelligence Memo 207, M.I.T. Project MAC (August 1970).

15. Ianov, Iu. The logical schemes of algorithms (Russian). *Problems of Cybernetics*, 1 (1958) 75–127. English translation: Pergamon Press Ltd., 1960, pp. 82–140.

16. Landin, Peter J. The mechanical evaluation of expressions. *The Computer Journal*, 6, 4 (January 1964).

17. Luckham, David C., Park, David M. R., and Paterson, Michael S. On formalised computer programs. *Journal of Computer and System Sciences*, 4, 3 (June 1970) 220–249.

18. Manna, Zohar. Program schemas. In Aho, Alfred V., editor, *Currents in the Theory of Computing*, Prentice-Hall, Inc. (1973) 90–142.

19. Mazurkiewicz, Antoni W. Proving algorithms by tail functions. *Information and Control*, 18, 3 (April 1971) 220–226.

20. McCarthy, John. A basis for a mathematical theory of computation. In Braffort, P. and Hirschberg, D., editors, *Computer Programming and Formal Systems*, North-Holland, Amsterdam (1963).

21. McCarthy, John *et al. The LISP 1.5 Programmers Manual*. M.I.T. Press (1963).

22. Moon, David A. *MACLISP Reference Manual, revision 0.* Project MAC—M.I.T., Cambridge, Massachusetts (April 1974).

23. Morris, F. Lockwood. The next 700 programming language descriptions. *Lisp and Symbolic Computation* (1993). Appears in this issue. Original manuscript dated November 1970.

24. Morris, Jr., James H. A bonus from van Wijngaarden's device. *Communications of the ACM*, 15, 8 (August 1972) page 773.

25. Moses, Joel. The function of FUNCTION in LISP (or, why the FUNARG problem should be called the environment problem). *SIGSAM Bulletin*, 15 (July 1970).

26. Paterson, Michael S. *Equivalence Problems in a Model of Computation.* PhD thesis, Cambridge University (August 1967). Also appears as Artificial intelligence Technical Memo No. 1, M.I.T. A.I. Lab, Cambridge, Massachusetts, issued November 1970.

27. Paterson, Michael S. Program schemata. In Michie, D., editor, *Machine Intelligence 3*, chapter 2, Edinburgh University Press (1968) 18–31.

28. Paterson, Michael S. and Hewitt, Carl E. Comparative schematology. In *Record of Project MAC Conference on Concurrent Systems and Parallel Computation*, ACM (December 1970) 119–127.

29. Randell, B. and Russell, L. J. *Algol 60 Implementation*. Academic Press, New York (1964).

30. Reynolds, John C. Definitional interpreters for higher order programming languages. In *Proceedings of the ACM Annual Conference, Boston* (August 1972) 717–740.

31. Reynolds, John C. The discoveries of continuations. *Lisp and Symbolic Computation* (1993). Appears in this issue.

32. Rutledge, J. On Ianov's program schemata. *Journal of the ACM*, 11, 1 (January 1964) 1–9.

33. Sabry, Amr and Felleisen, Matthias. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation* (1993). Appears in this issue.

34. Steele Jr., Guy L. *Common LISP*. Digital Press, second edition (1990).

35. Strachey, Christopher and Wadsworth, Christopher P. *Continuations, a Mathematical Semantics for Handling Full Jumps*. Technical Monograph PRG–11, Oxford University Computing Laboratory (January 1974).

36. Strong, H. Raymond. Translating recursion equations into flow charts. *Journal of Computer and System Sciences*, 5, 3 (1971) 254–285.

37. van Wijngaarden, Adriaan. Recursive definition of syntax and semantics. In T. B. Steel, Jr., editor, *Formal Language Description Languages for Computer Programming*, North-Holland, Amsterdam (1966) 13–24.