LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

An Introduction to the Lambda Calculus

Computation Structures Group Memo 258
July 1984

Keshav Pingali
Vinod Kathail

This paper was prepared for Summer Session course 6.83s.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Table of Contents

List of Figures

An Introduction to the Lambda-Calculus

1. Background

There are two ways of looking at functions - as a set of ordered pairs of argument and value (sometimes called the *graph* of the function) or as an encoding of a process for going from argument to value. The first approach is used when one is interested only in *defining* a function while the second approach is used when one is interested in *computing* the function. The second viewpoint is strictly weaker than the first because we can define functions which are not computable; nevertheless, this notion of functions is more useful in computer science. These notes are a summary of the λ-calculus of Church which is one particular way of encoding computable functions.[1]

Some notion of functions is present in most imperative programming languages such as FORTRAN or ALGOL. However, functions and procedures in these languages are quite different from functions in the mathematical sense because the invocation of a function in such languages is permitted to cause "side-effects". Moreover, these languages place irksome restrictions on how one can use functions. For example, in most programming languages, it is impossible to write a function without giving it a name. Other languages will not permit functions to be arguments of functions while others will not permit the programmer to return functions as results of function applications.

In the λ-calculus, functions are "first-class" objects in the sense that it is possible to express them without giving them a name, and they can be passed to and returned from other functions. In order to permit this flexibility, it is necessary to introduce an operation called *abstraction*. A simple example illustrates the need for this operation. Consider the integer *2* and the function *always-two*, which when applied to any integer, returns 2. We would like to regard these two objects as being different - for example, applying the integer 2 to the integer 5 should result in an error while always-two when applied to 5 should return 2. Therefore, we need some syntax to distinguish the integer 2 from the function always-two. We might do this by writing

```
define always-two(x) = 2
```

but this forces us to name the function. The notion of *abstraction* permits an elegant solution to the problem. The function *always-two* is defined as (λx:integer. 2) where the intended meaning of this syntax is exactly that which was expressed informally in English in talking about the always-two function - *i.e.*, the expression (λx : integer. 2) denotes a function of one integer-valued variable, which always returns 2. This operation of forming a function from an expression such as 2 or (x + 1) is called *abstraction*. When the domain of the argument is obvious from the context, it may be dropped. Thus, we could have

---

[1] The intent here is not so much to present the λ-calculus as a formal mathematical system but to summarize the basic ideas of the λ-calculus while appealing to the reader's intuitions about programming languages and computation. The reader who is interested in a more complete exposition is referred to Stoy's book on denotational semantics [4]. An encyclopedic reference for just about any noteworthy result on the λ-calculus is Barendregt's book on the lambda-calculus [1] - while this book is very well-written, it is not intended for the casual reader.

expressed the always-two function as (λx.2). The same idea can be used to define more complicated functions. For example, we can write (λx. (x + 1)) to denote the successor function. The operation of applying a function say (λx. M) to an argument N (where M and N are expressions) is expressed as ((λx. M) N). Thus, ((λx. (x + 1)) 2) denotes a function application. The result of this application may be found by "substituting" 2 for all occurrences of x in the expression (x + 1) and simplifying the resulting expression.

To make these notions rigorous, we need to be precise about what we mean by "all occurrences of a variable" in an expression or by "substitution". This is done in in section 2, where we describe the language of the λ-calculus and define three conversion rules called the α-, β- and η- rules which describe how expressions in the λ-calculus can be transformed. Section 3 is a brief description of some results about conversion. The important result presented in this section is the Church-Rosser theorem. In section 4, we demonstrate the power of the λ-calculus by showing how integers, booleans, data structures, and operations such as addition, test-for-zero etc. can be encoded in the λ-calculus. This section will establish the close correspondence between functional languages and the λ-calculus, thereby justify the claim that functional languages are "syntactic sugaring" for the λ-calculus. In Section 5, we introduce the theory of combinators and show its relation to the λ-calculus. The last section contains some additional concepts and results (in particular, the lemma of parallel moves) about the λ-calculus.

2. Lambda-terms and Conversion

2.1. Syntax of Lambda-terms

Expressions of the λ-calculus are words defined on the alphabet { λ, . , (,)} and a countably infinite set of variables (say {a, b, c, a0, a1 etc.}).

Definition 1: A λ-term (also known as a well-formed expression or wfe) is an expression that is constructed according to the following BNF :

<wfe> ::= <variable>
| (<wfe> <wfe>)
| (λ<variable>.<wfe>)

In definitions, we will let *x* and *y* be meta-variables that stand for any variable. ε(adorned if necessary with subscripts) and capital letters such as M and A will be meta-variables that stand for any wfe. We will write x $\equiv$ y to denote that x and y stand for the *same* variable.

Definition 2: A wfe (λx.ε) is called a λ-*abstraction* in which x is the *bound variable* (or *bv*) and ε is the *body*. A wfe (ε_1 ε_2) is called an *application* in which ε_1 and ε_2 are the *rator* and *rand* respectively.

For example, ((λx.(λy.(x y))) z) is an application in which the rator is (λx.(λy.(x y))) and the rand is z. The bound variable of the λ-abstraction (λx.(λy.(x y))) is x and the body is (λy (x y)).

To avoid writing too many parentheses, we adopt the usual λ-calculus conventions. We

assume that application is left-associative; so $\varepsilon_1\ \varepsilon_2\ \ldots\ \varepsilon_n$ stands for $((\ldots(\varepsilon_1\ \varepsilon_2)\ldots\ \varepsilon_n))$. Moreover, when a λ-abstraction occurs in a wfe, the body of the λ-abstraction is taken as extending as far as possible - that is to the first unmatched closing parenthesis or to the end of the wfe, whichever occurs first. For example, (λx.λy. x y) z stands for ((λx.(λy.(x y))) z)

2.2. Substitution

To define substitution rigorously, we need the concept of free and bound variables in wfe's. Intuitively, a free occurrence of a variable x in a wfe ε is the occurrence of x which is not part of a λ-abstraction of the form λx.M. For example, the occurrence of z in ((λx.λy. x y) z) is free whereas the occurrences of x and y are not. In the last section, we will give a precise way of identifying a particular occurrence of a variable (in general, any wfe) in a wfe; for now, we will assume that it is clear from the context.

Definition 3: A *free* occurrence of a variable in a wfe is defined inductively as follows -

1. the occurrence of x in the wfe x is free
2. a free occurrence of a variable x in ε_1 or ε_2 is free in the application $(\varepsilon_1\ \varepsilon_2)$
3. an occurrence of a variable x in a λ-abstraction $\lambda y.\varepsilon$ is free iff $x \neq y$ and x is free in ε.

A variable occurrence which is not free is said to be *bound*.

Note that a variable can be both free and bound in a wfe. For example, in the wfe (x (λx.x)), the first occurrence of *x* is free while the second occurrence of *x* is bound.

We now define the substitution operation $\varepsilon_1[\varepsilon_2/x]$ whose informal meaning is "replace all free occurrences of x in ε_1 by ε_2". As is well-known, care must be taken to avoid "capture of free variables". Consider, for example, the substitution of (λz.y) for x in (λy.x y (λy.y)). A naive way of performing the substitution may produce the wfe (λy. (λz.y) y (λy.y)) where the free occurrence of y in the wfe (λz.y) has gotten bound in the wfe resulting from substitution. It can be shown that this causes logical difficulties, and renders the system inconsistent. In fact, in the early days of the λ-calculus, there were several incorrect attempts to define substitution that were accepted by logicians for a time until those definitions were shown to lead to inconsistencies ! The first correct definition of substitution was given by Curry and is given below -

Definition 4: $\varepsilon_1[\varepsilon_2/x]$ is defined inductively as follows -

$$y[\varepsilon_2/x] = \varepsilon_2 \text{ if } x \equiv y;\ y \text{ otherwise}$$

$$(\varepsilon_3\ \varepsilon_4)[\varepsilon_2/x] = (\varepsilon_3[\varepsilon_2/x]\ \ \varepsilon_4[\varepsilon_2/x])$$

$$(\lambda y.\varepsilon_3)[\varepsilon_2/x] = (\lambda y.\varepsilon_3) \text{ if } (x \equiv y);\ (\lambda z.((\varepsilon_3[z/y])\ [\varepsilon_2/x])) \text{ otherwise,}$$

where z is a variable which does not occur in ε_2 *or* ε_3

The capturing of the free variables of ε_2 is avoided in the third clause by *renaming* the bound variable of the λ-abstraction (*i.e.*, changing $\lambda y.\varepsilon_3$ to $\lambda z.(\varepsilon_3[z/y])$). As we shall see later, the renaming operation is one of the rules of conversion in the λ-calculus.

Let us consider the example introduced earlier in this section, *i.e.* substitution of (λz.y) for x in the wfe (λy.x y (λy.y)).

$$
\begin{aligned}
(\lambda y.x\ y\ (\lambda y.y))[(\lambda z.y)/x] &= \lambda p.(((x\ y\ (\lambda y.y))[p/y])[(\lambda z.y)/x]) && \text{(by the third clause)}\\
&= \lambda p.((x\ p\ ((\lambda y.y)[p/y]))[(\lambda z.y)/x]) && \text{(by the first two clauses)}\\
&= \lambda p.((x\ p\ (\lambda y.y))[(\lambda z.y)/x]) && \text{(by the third clause)}\\
&= \lambda p.(\lambda z.y)\ p\ (\lambda y.y)
\end{aligned}
$$

Notice that the occurrence of y in the expression (λy.y) is not a free occurrence of y in the body of the outermost λ-abstraction, and thus, p is not substituted for y in the expression (λy.y).

As can be readily appreciated, substitution is a rather complicated notion because care must be taken to distinguish between free and bound variables and to avoid capture of free variables. These complications have led some logicians to work with alternative systems called *combinatory systems*. A *combinator* is a λ-expression with no free variables. By working with combinators, it is possible to define abstraction without having to work with bound variables, thereby avoiding the problems of substituting for variables completely. We will describe combinatory systems more fully in section 5.

2.3. Conversion

We now have all the tools to describe the process of computation in the λ-calculus. The basic idea is to transform a wfe into other wfe's by using certain simplification rules called *conversion rules* so as to generate a wfe that cannot be simplified any further. This process is called *conversion*, and is very similar to the way we evaluate arithmetic expressions (for example, those built up from the integers and the addition and multiplication operations). Consider the arithmetic expression ((2 + 3) * (4 + 2)). This expression can be simplified by using the rules for addition and multiplication of integers to generate the expression (5 * (4 + 2)) by performing the first addition, and then performing the addition to generate the expression (5 * 6) which can be further simplified to 30. Thus, by using the simplification rules of arithmetic, we can generate a sequence of expressions such that the final expression cannot be simplified any further. While arithmetic expressions provide a useful intuition for understanding the conversion rules of the λ-calculus, the reader should be aware that since the λ-calculus is a far more expressive system than arithmetic expressions, it is possible to define wfe's which cannot be simplified to yield a wfe that cannot be simplified any further.

We now define the so-called α-, β- and η- conversion rules of the λ-calculus. We write ε_1 **cnv** ε_2 (to be read as "ε_1 is convertible to ε_2") to indicate that either of the wfe's may be replaced by the other whenever one of them occurs as an expression or as a subexpression of a larger expression.

Definition 5: The conversion rules of the λ-calculus are the following -

α-conversion - If x_1 is not free in ε_1, then $\lambda x.\varepsilon_1$ **cnv** $\lambda x_1.\varepsilon_1[x_1/x]$.

β-conversion - $(\lambda x.\varepsilon_1)\varepsilon_2$ **cnv** $\varepsilon_1[\varepsilon_2/x]$.

η-conversion - If x doesn't occur free in ε, then $(\lambda x.\varepsilon x)$ **cnv** ε.

Of these three rules, we have already seen α-conversion when we described substitution. Indeed, the rule of α-conversion merely says that the bound variables of a wfe can be renamed without changing the "meaning" of the wfe. For this reason, many authors do not elevate it to the status of a conversion rule, and prefer to identify α-convertible wfe's at the syntactic level. We have already hinted at the β-conversion rule in the introduction when we discussed λ-notation and its relation to functions. The β-conversion rule asserts that either of $(\lambda x.\varepsilon_1)\varepsilon_2$ or $\varepsilon_1[\varepsilon_2/x]$ can be replaced by the other in any wfe. In practice, one is usually interested only in replacing $(\lambda x.\varepsilon_1)\varepsilon_2$ by $\varepsilon_1[\varepsilon_2/x]$, a process which is called *β-contraction* or *β-reduction*. In other direction, the replacement of $(\lambda x.\varepsilon_1)\ \varepsilon_2$ by $\varepsilon_1[\varepsilon_2/x]$ is known as *β-expansion*. In programming languages terms, the intuitive content of β-reduction is essentially that of function application - *i.e.*, to apply a function to an argument, one can replace all occurrences of the formal parameter of the function in the body of the function by the actual parameter, subject to renaming of bound variables to avoid name conflicts. η-conversion can be thought of an assertion about the behavior of wfe's under application - we will not deal with this rule any further in these notes, and it is included here only for completeness.

The only simplification rule we will consider is the β-reduction[2]. A wfe of the form $((\lambda x.\varepsilon_1)\ \varepsilon_2)$ will be called a β-redex or simply a *redex*, and the wfe $\varepsilon_1[\varepsilon_2/x]$ will be called the *contractum* of the redex. The operation of replacing an occurrence of a redex in any wfe with its contractum will be called *contracting the redex*. We will write $\varepsilon_1 \rightarrow \varepsilon_2$ to indicate that ε_2 is obtained from ε_1 by contracting some redex which is a sub-expression of ε_1.

A *reduction sequence* is a sequence (finite or infinite) of wfe's ε_0, ε_1, ε_2 ... such that for each i, ε_{i+1} is obtained by contracting a redex in ε_i. The *initial wfe* of the reduction sequence is ε_0. We will write this reduction sequence as

$$\varepsilon_0 \rightarrow \varepsilon_1 \rightarrow \varepsilon_2 \ldots.$$

We will write "ε_1 **red** ε_2" (read as ε_1 reduces to ε_2) to indicate that there is a reduction sequence whose initial wfe is ε_1 and which terminates at ε_2. In same spirit, we extend the meaning of ε_1 **cnv** ε_2 to indicate that ε_1 can be converted to ε_2 by repeatedly using the β-conversion rule.

Consider, for example, the following wfe's:

$\varepsilon_{cons} \equiv (\lambda x.\lambda y.\lambda f.f\ x\ y)$
$\varepsilon_{car} \equiv \lambda f.f\ (\lambda x.\lambda y.x)$,
$\varepsilon_{cdr} \equiv \lambda f.f\ (\lambda x.\lambda y.y)$.

Then, the wfe $(\varepsilon_{car}\ (\varepsilon_{cons}\ a\ b))$ reduces to *a* as shown by the following reduction sequence in which the redex contracted at each step is the underlined redex.

---

[2] In the next section, we will show that β-reduction is sufficient if we are interested only in "simplifying" a wfe.

$(\varepsilon_{car}\ (\varepsilon_{cons}\ a\ b))$ $\equiv(\lambda f.f(\lambda x.\lambda y.x))((\lambda x.\lambda y.\lambda f.f\ x\ y)\ a\ b)$
$\rightarrow (\lambda f.f(\lambda x.\lambda y.x))((\lambda y.\lambda f.f\ a\ y)\ b)$
$\rightarrow (\lambda f.f(\lambda x.\lambda y.x))(\lambda f.f\ a\ b)$
$\rightarrow (\lambda f.f\ a\ b)(\lambda x.\lambda y.x)$
$\rightarrow (\lambda x.\lambda y.x)\ a\ b$
$\rightarrow (\lambda y.a)\ b$
$\rightarrow a$

Notice that the final wfe, *i.e. a*, cannot be reduced any further. The reader is urged to verify that $(\varepsilon_{cdr}\ (\varepsilon_{cons}\ a\ b))$ **red** *b* .

A wfe is said to be *in normal form* if it does not contain a redex as a sub-expression. A wfe is said to *have* a normal form if there is a wfe in normal form to which it is reducible. As we mentioned earlier, computing the meaning of arithmetic expressions involves applying the simplification rules of arithmetic until we get an expression which cannot be simplified any further. At the risk of over simplification, the same is true for the λ-calculus as well - the process of computation in the λ-calculus can be thought of as repeatedly applying the conversion rules (as far as we are concerned, just β-reduction) until we get a wfe that cannot be simplified any further (*i.e.*, a wfe in normal form). The over simplification arises from the fact that not all wfe's have normal forms. For example, the wfe (λx. x x) (λx. x x) does not have a normal form, nor does the so-called paradoxical combinator of Curry (λf.(λy .f(y y)) (λy.f(y y))) . Wfe's without normal forms correspond roughly to non-terminating computations in conventional programming languages. However, not all wfe's without normal forms can be thought of as "meaningless" in the sense of representing the completely undefined computation. A deeper discussion of this topic is beyond the scope of these notes, and the interested reader is referred to the paper by Wadsworth on this subject[3] [7].

Having accepted the fact that a wfe need not have a normal form, the reader may wonder if it possible for a wfe to have two different normal forms (modulo α-conversion). Since a wfe can have more than one redex in it, is it possible that two different orders of performing redexes results in two different normal forms ? If this were so, our assertion that λ-calculus models the behavior of functions and that normal forms correspond roughly to the "result of a computation" would be questionable. We will show in the next section that this is not so.

3. Some Important Results about Convertibility

When dealing with arithmetic expressions, we know that the order in which simplification rules are applied is unimportant, and every order of applying these rules will terminate and produce the same answer. This property is sometimes called the *strong Church-Rosser*

---

[3] Intuitively, we recognize that there is a difference between a program that diverges without producing any output, and a program that diverges after producing some output. The λ-calculus analog of a program that diverges after producing some output is a wfe which does not have a normal form but does have a so-called *head-normal form*. A program which diverges without producing any output at all corresponds roughly to a wfe *without* a head-normal form. For this reason, not all wfe's without normal forms can be thought of as representing ⊥ (the purely diverging computation).

property. What about the λ-calculus ? It can be shown that any two sequences of reductions of a wfe that produce normal forms must produce the *same* normal form (modulo α-conversion). Unlike the case for arithmetic expressions, however, there may be some sequences of reductions which do not produce a normal form at all. Consider, for example, the wfe

$\varepsilon \equiv ((\lambda y.z)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)))$.

This wfe has a normal form z. However, the reduction sequence

$\varepsilon \rightarrow \varepsilon \rightarrow \varepsilon$ .....

in which the redex contracted at every step is ((λx.x x) (λx.x x)) does not terminate. Thus, for any wfe, every terminating reduction sequence produces the same normal form but not every reduction sequence terminates. Moreover, for any wfe that has a normal form, a wfe generated at any step of a non-terminating reduction sequence can be reduced to the normal form by some reduction sequence. This property is called the *weak Church-Rosser property* and the theorem which asserts that the λ-calculus has this property is stated below.

Theorem 6: (Church-Rosser Theorem I)

If ε_1 **cnv** ε_2, then there exists a wfe ε_3 such that ε_1 **red** ε_3 and ε_2 **red** ε_3.

To understand the theorem, it is helpful to visualize the conversion of ε_1 to ε_2 as a path from ε_1 to ε_2 in which downward sloping lines represent β-reductions and upward lines represent β-expansions. The path A in Figure 3-1 shows the conversion of ε_1 to ε_2 in five steps. The Church-Rosser theorem states that ε_1 can be converted to ε_2 in a way so that no expansion step precedes any reduction step - see path B in Figure 3-1.

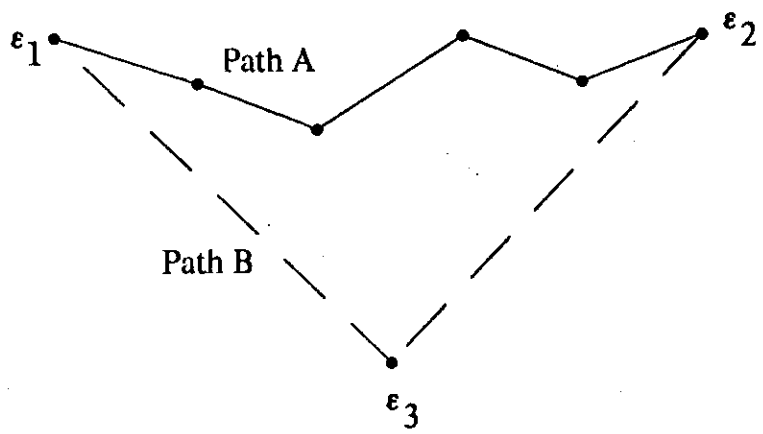

Figure 3-1: The Church-Rosser Theorem

Corollary 1: If ε_1 **cnv** ε_2 and ε_2 is in the normal form, then ε_1 **red** ε_2.

Corollary 2: If a wfe has a normal form, then the normal form is unique up to α-conversion.

The first corollary states that β-reduction is sufficient to reduce wfe's to their normal forms - we need not consider β-expansion.

The second corollary states the weak Church-Rosser property of the λ-calculus mentioned earlier in the section. To see that the corollary follows from the theorem, suppose a wfe has two normal forms, say N_0 and N_1 such that N_0 is not α-convertible to N_1. By going from N_0 to the original wfe and then to N_1, we can convert N_0 to N_1. Hence, by the Church-Rosser theorem, there must exist a wfe to which both N_0 and N_1 are reducible. However, this is not possible as N_0 and N_1 are in normal form.

From the Church-Rosser theorem, it follows that one can always find the normal form of a wfe that has one by trying all possible reductions for it. However, there is a more direct way in the sense that we need to try just one particular sequence which we will now define.

Definition 7: For a redex R = $(\lambda x.M)A$, the occurrence of λx is called the *head* of R. If R and S are two redexes in the same wfe ε, then R is said to be to the left of S if the head of R lies to the left of of the head of S in the linear representation of ε. For any wfe not in normal form, there is a unique redex R which is to the left of all other redexes in the wfe; this redex is called the *leftmost* redex of the wfe.

For example, in the wfe $((\lambda x.(\lambda y.y)\ x)\ ((\lambda z.z)\ w))$, the redex $((\lambda y.y)\ x)$ lies to the left of the redex $((\lambda z.z)\ w)$. The leftmost redex in the wfe is the wfe itself.

Definition 8: Normal Order Reduction

Let

$$\varepsilon_0 \rightarrow \varepsilon_1 \rightarrow .... \rightarrow \varepsilon_n$$

be a reduction of ε_0 to ε_n, and for each i, let ε_{i+1} be the result of contracting redex R_i in ε_i. This reduction sequence is called a *normal order reduction* iff for every i, R_i is the leftmost redex in ε_i.

Theorem 9: (Church-Rosser Theorem II)

If ε_1 **red** ε_2, and ε_2 is in normal form, then there exists a normal order reduction from ε_1 to ε_2.

From a computational viewpoint, this is a very satisfying result - if a wfe has a normal form, then the normal form can be computed by performing a normal order reduction on the wfe. Of course, we cannot tell *a priori* whether a wfe has a normal form or not, because that would be equivalent to solving the halting problem, but if it does, then normal order evaluation will compute it.

A different order of evaluation is the so-called *applicative order* reduction in which the rator and rand of a redex are reduced separately to normal form before the application is performed. Let us define the *leftmost innermost* redex in a wfe to be the leftmost redex that

doesn't contain any other redex. Then, applicative order reduction can be defined as follows:

Definition 10: Applicative Order Reduction

Let

$\varepsilon_0 \rightarrow \varepsilon_1 \rightarrow .... \rightarrow \varepsilon_n$

be a reduction of ε_0 to ε_n, and for each i, let ε_{i+1} be the result of contracting redex R_i in ε_i. This reduction sequence is called an *applicative order reduction* iff for all i, R_i is the leftmost innermost redex in ε_i where the leftmost innermost redex is defined to be the leftmost redex that contains no other redex.

Consider the wfe $\varepsilon \equiv ((\lambda y.z)\ ((\lambda x.x\ x)\ (\lambda x.x\ x)))$ which has the normal form z. Applicative order reduction of ε will contract the redex $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$ at each step, and therefore, will never terminate. Thus, there are wfe's for which normal order reduction terminates while applicative order reduction may not. However, in practice, it is found that when applicative order reduction terminates, it usually does so in fewer steps than the corresponding normal order reduction. For example, consider the wfe $((\lambda x.x\ x)\ ((\lambda x.x)\ a))$. Applicative order reduction yields the normal form in two steps while normal order reduction takes three steps. For this reason, most practical interpreters for programming languages perform applicative order reduction. Normal order reduction will not evaluate an argument if it is not needed to produce the normal form - however, it may end up evaluating the argument many times. The power of normal order evaluation has its price. For a discussion on attempts to make normal order evaluation more "efficient", see [6].

4. Connections with Functional Languages

In this section, we show that integers and functions on integers such as the successor function, addition, and test-for-zero can be represented in the λ-calculus. We then introduce the so-called paradoxical combinator of Curry and show how it can be used to define recursive functions. By extending the results presented here (as in Chapters 6 and 8 of Barendregt), it is possible to show that the class of functions that can be defined in the λ-calculus is exactly the class of partial recursive functions. We also point out how to represent data structures in the λ-calculus.

There are many possible representations of integers in the λ-calculus. The one we will present here was first suggested by Church, and is commonly known as the Church representation of the integers. The Church representation of an integer N will be denoted by [N] (this notation is somewhat non-standard - the usual notation is N with a bar on top as in N, but we are constrained by the text-formatter).

Recall that a *partial numeric function* is a partial mapping $\varphi: N^p \rightarrow N$ for some $p \in N$. We now define precisely what it means for a partial numeric function to be λ-definable.

Definition 11: A partial numeric function of *p* arguments is said to be λ-definable iff there exists a wfe ε such that for all $n_1, n_2 \ldots n_p \in N$

if $\varphi(n_1, n_2, \ldots n_p)$ is defined

then $\varepsilon\,[n_1]\,[n_2] \ldots [n_p] = [\varphi(n_1, n_2, \ldots n_p)]$
else $\varepsilon\,[n_1]\,[n_2] \ldots [n_p]$ has no normal-form.

The Church representation of integers is the following -

$[0] \equiv \lambda x.\lambda y.y$
$[1] \equiv \lambda x.\lambda y.(x\ y)$
$[2] \equiv \lambda x.\lambda y.(x(x\ y))$
.....
$[n] \equiv \lambda x.\lambda y.(x(x \ldots (x\ y) \ldots))$

In other words, the representation of the integer *n* is a λ-abstraction that applies its first argument to its second argument *n* times.

The successor function can be λ-defined by the wfe

$\text{Suc} \equiv \lambda x.\lambda y.\lambda z.(y(x\ y\ z))$.

To verify this consider the application of Suc to [2].

$$\begin{aligned}\text{Suc}\ [2] &\equiv (\lambda x.\lambda y.\lambda z.(y(x\ y\ z)))\ (\lambda x.\lambda y.(x\ (x\ y))))\\ &\rightarrow (\lambda y.\lambda z.(y\ ((\lambda x.\lambda y.(x\ (x\ y)))\ y\ z)))\\ &\rightarrow (\lambda y.\lambda z.(y\ ((\lambda p.(y\ (y\ p)))\ z)))\\ &\rightarrow (\lambda y.\lambda z.(y\ (y\ (y\ z))))\\ &\equiv [3]\end{aligned}$$

By using the fact that the representation of *n* is a λ-abstraction that applies its first argument to its second argument *n* times, the reader can verify that *addition* can be λ-defined as follows -

$\text{plus} \equiv \lambda x.\lambda y.(x\ \text{Suc}\ y)$

where suc is the λ-representation of the successor function that was defined earlier. The λ-definitions of *times* and *exponentiation* are left to the reader.

Boolean values can be represented as follows -

$\text{True} \equiv \lambda x.\lambda y.x$
$\text{False} \equiv \lambda x.\lambda y.y$

With this representation, the test-for-zero function of one argument which returns true if the argument is 0 and false otherwise can be λ-defined to be

$\text{IsZero} \equiv \lambda n.\ (n\ (\text{True}\ \text{False})\ \text{True})$.

Using the functions λ-defined so far, the reader should be able to λ-define many common numeric functions such the Predecessor function. However, in order to λ-define functions that are defined recursively (for example, the factorial function), a new trick is needed. Consider the definition -

```
define fact n = if n = 0 then 1
                         else n*fact(n-1)
```

Using the functions we have defined so far, we can express this as follows -

Fact = λn. IsZero n [1] (Times n (Fact(Pred n)))

where Pred is the λ-definition of the Predecessor function.

The form on the right hand side of this equation does not define the factorial function because there is an occurrence of *Fact* in it; rather, the form shown above should be thought of an equation one of whose solutions is the λ-definition of Fact that we seek. The standard way to solve such equations is to write the equation as follows -

Fact = H Fact
where H ≡ λf.λn. IsZero n [1] (Times n (f (Pred n)))

In other words, the solution we seek is a fixed point of the *H* function. To find the solution, we use the following standard theorem of the λ-calculus.

Theorem 12: For any wfe ε_1, there exists a wfe ε_2 such that $(\varepsilon_1\ \varepsilon_2) = \varepsilon_2$.
Proof: Let ε_2 be (Y ε_1) where Y ≡ (λf.(λx.(f (x x))) (λx.(f (x x)))).

The combinator Y is sometimes called the *paradoxical* combinator. Using this combinator, the λ-definition of the factorial function is

Y λf.λn. IsZero n [1] (Times n (f (Pred n)))

The reader is urged to use this expression to compute (Fact 2) in order to get a feel for how the Y combinator works.

Now we consider how data structures can be represented in the λ-calculus. For the purpose of this section, we will assume that a data structure is completely characterized by a set of functions (*e.g.*, constructor and selector functions) and a set of axioms which describe the behavior of the functions. Suppose D is a data structure characterized by the set of functions $\{f_1, f_2, \ldots f_n\}$ and the set of axioms A. If we can find wfe's $\varepsilon_1, \varepsilon_2, \ldots \varepsilon_n$ such that they satisfy the axioms in the set A provided we interpret each f_i as ε_i and equality as β-convertibility, then we will say that the data structure D can be represented in the λ-calculus using the wfe's $\varepsilon_1, \varepsilon_2, \ldots \varepsilon_n$.

As an example, suppose we want to represent lisp-like data structure, called *pair*, in the λ-calculus. The constructor and selector functions for the data structure *pair* are usually called *cons*, *car* and *cdr*, and they satisfy the following axioms:

(*car* (*cons* x y)) = x ;
(*cdr* (*cons* x y)) = y .

Let us consider the wfe's ε_{cons}, ε_{car} and ε_{cdr} defined in Section 2. In that section, we showed that (ε_{car} (ε_{cons} a b)) **red** *a* and (ε_{cdr} (ε_{cons} a b)) **red** *b*. Since these wfe's satisfy the same axioms as the functions *cons*, *car* and *cdr*, they can be used to represent the data structure *pair* in the λ-calculus.

5. Combinatory Systems

The complications involved in working with variables (in particular, bound variables) have led to an alternative theory of functions known as combinatory systems. In this section, we define one such combinatory system.

Let us consider a system with one operation *application* and two objects *S* and *K*. Expressions in this system can be built from the two objects by repeated application. We will call these expressions *combinatory expressions*. With *S* and *K*, we associate two simplification rules -

(S f g x) = ((f x) (g x))
(K x y) = x

Notice that these two objects can be λ-defined as follows -

S = (λx.(λy.(λz. (x z) (y z))))
K = (λx.(λy.x))

Thus, any combinatory expression can be expressed in the λ-calculus. It is easy to show that any *closed* wfe of the λ-calculus can be expressed as a combinatory expression. Let I denote the combinatory expression (S K K) and verify that for any combinatory expression E, (I E) can be simplified to E by using the simplification rules for the S and K combinators. By using the λ-definitions of the S and K combinators, one can show -

1. $\lambda x.x = I$

2. $\lambda x.\varepsilon = K\,\varepsilon$ *if x is not free in* ε

3. $\lambda x.\varepsilon_1\,\varepsilon_2 = S\,(\lambda x.\varepsilon_1)\,(\lambda x.\varepsilon_2)$

By using these rules, we can eliminate all the λ's in the wfe successively in favor of K and S. For example,

(λx.(λy.y x))
= (λx.(S I (K x)))
= S(K(SI))(S(KK)I)

Thus, one can define standard translations from closed wfe's of the λ-calculus to combinatory expressions and back. By permitting variables to occur in combinatory expressions, we can extend this translation to any wfe of the λ-calculus. Notice that even in this extended system, there are no *bound* variables.

Turner [5] has suggested that one way to implement functional languages is to translate programs in such languages into combinatory expressions and reduce the resulting combinatory expressions on a machine specially designed for that purpose. One problem with the standard translation is that the length of the combinatory expression resulting from a wfe can be much larger than the length of the wfe. By introducing additional combinators, Turner has shown that this blow-up problem may be overcome. For a different approach to implementing functional languages using combinatory reduction, see [2].

6. Complete Reductions and Lemma of Parallel Moves

In this section, we describe some additional concepts and results in the λ-calculus. They are useful if one is interested in pursuing the subject matter, specially the properties of the β-conversion, in more detail. The Church-Rosser theorem can be proved easily using the lemma of parallel moves which is stated at the end of the section.

6.1. The Tree Representation of wfe's

Wfe's have an obvious tree structure - the tree structure of the wfe (λx.(λy.((+ ((* x) x)) ((* y) x)))) is shown in Figure 6-1. The tree structure of wfe's gives us a simple way of distinguishing two different occurrences of any wfe inside a wfe. The intuitive idea is to specify the path that must be followed in the tree in order to reach the root node of the occurrence of the wfe that we are interested in. More formally, a *selector* is either *rator*, *rand* or *body*. A *path* is a list of zero or more selectors. A path can be considered to be a function which when applied to a wfe, returns the wfe whose root node is obtained by starting at the root cell of the tree of the wfe and following the specified selectors. For example, the path *body.body.rator.rand.rand* applied to the tree of Figure 6-1 returns the wfe *x* and the path *body.body.rand* returns the wfe ((* y) x). If a path *p* does not exist in the graph, then the result of applying the function *p* to the graph is undefined. A sub-expression (*i.e.*, a particular occurrence of a wfe) of a wfe ε is a pair <p, ε_1> consisting of a path and a wfe ε_1 such that $p(\varepsilon) = \varepsilon_1$.

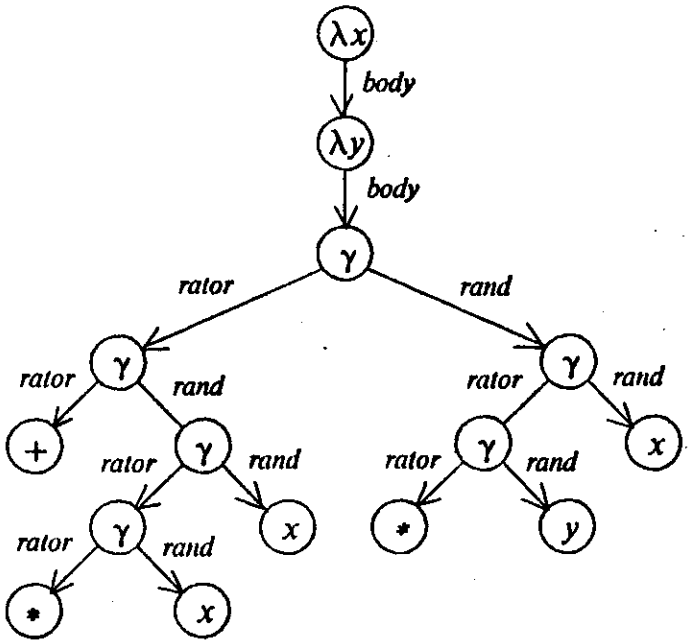

Figure 6-1: The Tree Representation for λ-expressions

6.2. Descendants and Ancestors

We now describe the notion of descendants and ancestors that is very useful for associating parts of wfe's in a reduction sequence with parts of the initial wfe of the sequence, and vice versa. These notions were defined by Morris [3] as a generalization of *residuals* as defined by Church and Curry.

Suppose $\varepsilon_1 \rightarrow \varepsilon_2$. A function *father* from parts of ε_2 to parts of ε_1 will be defined such that every part of ε_2 has a unique father in ε_1. The relation *son* is the inverse of father. Intuitively, the sons of a sub-expression of ε_1 are all the copies of that sub-expression in ε_2. As we will see, some parts of ε_1 will not have any sons in ε_2 while others may have more than one son.

Definition 13: Ancestors and Descendants

If R = (λx.M A) is a redex in a wfe ε_1, let ε_2 be the wfe obtained by contracting R. Let T1 and T2 denote the tree representations of ε_1 and ε_2 respectively. If N is a sub-expression of ε_2, the **father** of N is the sub-expression of ε_1 that is defined as follows:

1. If N is not part of the contractum of R, then let p be the path from the root node of T2 to N. The father of N is p(T1).

2. If N is part of a copy of A, then let p be the path from the root node of the copy of A to N. The father of N is p(A).

3. If N is any other part of the contractum of R, then let p be the path from the root node of the contractum of R to N. The father of N is p(M).

The **son relation** is the inverse of the father function.

Figure 6-2 summarizes this definition. Notice that neither the redex being contracted nor any free occurrence of *x* in M have any sons. A sub-expression that is part of the rand *A* has *k* sons where *k* is the number of free occurrences of *x* in *M*.

If ε_f is the wfe that results from performing a sequence of reductions on a wfe ε, then the ancestor and descendant relations between sub-expressions of ε and ε_f are the natural extensions by transitivity of the father and son relations respectively. Note that a redex has a descendant then the descendant must be a redex. The descendants of a redex are sometimes called its *residuals*.

Definition 14: Complete Reductions

If $\mathfrak{R}$ is a set of redexes in a wfe ε, then a sequence of reductions on ε is said to be a *reduction relative to* $\mathfrak{R}$ if the redex contracted at each step is a descendant of a redex in $\mathfrak{R}$. Such a reduction sequence is a *complete reduction relative to* $\mathfrak{R}$ if the final wfe does not contain any descendants of redexes in $\mathfrak{R}$.

For example, consider the wfe (($\underline{(\lambda x.\ x\ b)\ (\lambda x.\ x\ x)}$) $\underline{((\lambda x.x)\ a)}$) and let $\mathfrak{R}$ be the set containg the two underlined redexes. A complete reduction relative to the redexes in $\mathfrak{R}$ results in the wfe (((λx. x x) b) a). If, on the other hand, $\mathfrak{R}$ was the singleton set containing just the redex ((λx. x b) (λx. x x)), then the result of performing a complete reduction with respect to this set of redexes is the wfe (((λx. x x) b) ((λx.x) a)). Notice that even if $\mathfrak{R}$ included all the

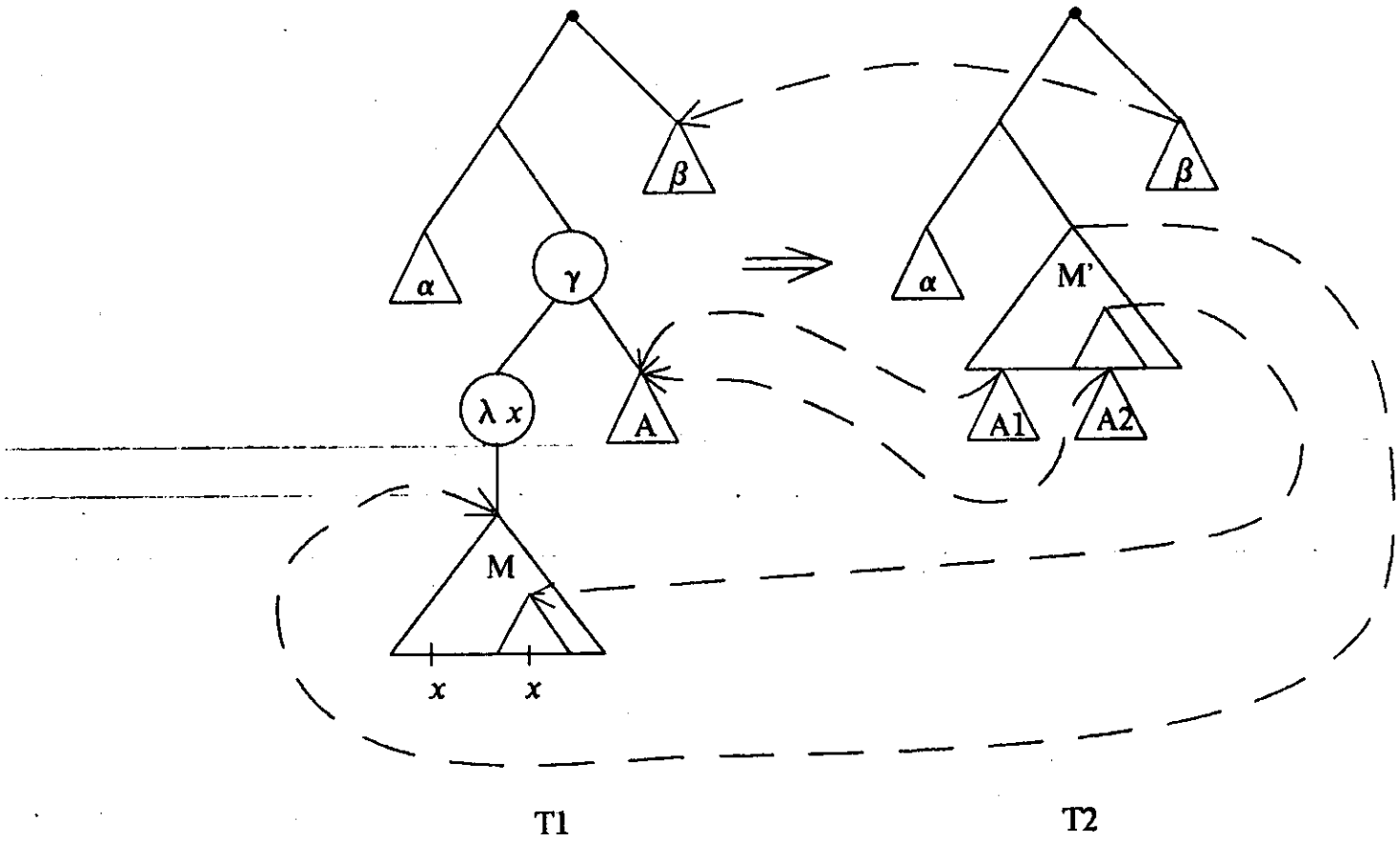

Figure 6-2: The Father Function in β-reduction

redexes in the original wfe, it is not necessary that the result of performing a complete reduction with respect to this set of redexes be a wfe in normal form because contracting a redex may create new redexes.

The reader may wonder what would happen if we performed a complete reduction with respect to a set of redexes in a wfe, thereby removing redexes created by contraction from consideration. Not surprisingly, we have the following result -

Lemma 15: Lemma of Parallel Moves

Let $\mathfrak{R}$ be a set of β-redexes in a wfe ε. Then

1. All reductions of ε relative to $\mathfrak{R}$ (including complete reductions) are finite.

2. Any reduction of ε relative to $\mathfrak{R}$ can be extended to a complete reduction of ε relative to $\mathfrak{R}$.

3. All complete reductions of ε relative to $\mathfrak{R}$ end with the same wfe, say ε_f. Further (Morris [3]), any sub-expression of ε_f has the same ancestor in ε

regardless of which reduction sequence is followed.

Note that this lemma is not true if η-redexes are included in $\mathfrak{R}$.

References

1. Barendregt, H.. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1981.

2. Johnsson, Thomas. Efficient Compilation of Lazy Evaluation. Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, Montreal, June, 1984, pp. 58-69.

3. Morris, J. Lambda-calculus models of programming languages. 57, Laboratory for Computer Science, MIT, Cambridge, Mass., 1968.

4. Stoy, J.. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* M.I.T. Press, 1977.

5. Turner, D. A. "A New Implementation Technique for Applicative Languages". *Software - Practice and Experience* , 9 (1979), 31-49.

6. Wadsworth, C.P.. *Semantics and Pragmatics of the Lambda-Calculus.* University of Oxford, , 1971.

7. Wadsworth, C.P. "The Relation Between Computational and Denotational Properties for Scott's D_{∞}-models of the Lambda-Calculus". *SIAM Journal of Computing* 5, 3 (September 1976), 488-521. .