Foundational Calculi for Programming Languages

[To appear in the *CRC Handbook of Computer Science and Engineering*]

Benjamin C. Pierce*

December 22, 1995

1 Introduction

In the mid 1960s, Landin observed that a complex programming language can be understood in terms of a tiny "core language" capturing the essential mechanisms of some programming, style together with a collection of convenient "derived forms" whose behavior is understood by translating them into the core (cf. [Tennent, 1981]). Landin's core language was the lambda-calculus, a formal system in which all computation is reduced to the basic operations of function definition and application. Since the 60s, the lambda-calculus has seen widespread use in the specification of programming language features, language design and implementation, and the study of type systems. Its importance arises from the fact that it can be viewed simultaneously as a simple programming language *in* which computations can be described and as a mathematical object *about* which rigorous statements can be proved.

The lambda-calculus has a strong claim to be a *canonical* model of purely functional computation (the programming paradigm where the only observable properties of an expression are its behavior when applied to arguments). Not only does it capture the ideas of function definition and application in a clear, intuitive way, but all other known models of functional computation — Turing Machines, general recursive functions, control structures such as *while* and *goto*, etc. — can be shown to describe exactly the same class of functions. (For a survey of these results, see [Davis, 1982].) For concurrent and distributed systems, no such canonical model has yet emerged. Instead, many different "process calculi" are being studied, each embodying some particular set of primitives for concurrent computation.

This chapter sketches the definitions and some basic properties of the lambda-calculus and a representative process calculus called the pi-calculus.

2 Lambda-Calculus

Procedural abstraction is a key feature of most programming languages. Instead of writing the same calculation over and over, we write a procedure or function that performs the calculation abstractly, in terms of one or more named parameters, which we instantiate as needed, providing values for the parameters in each case. For example, we might rewrite an expression like $(5 \cdot 4 \cdot 3 \cdot 2 \cdot 1) + (7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1) - (3 \cdot 2 \cdot 1)$ as $\mathit{Factorial}(5) + \mathit{Factorial}(7) - \mathit{Factorial}(3)$,

*University of Cambridge, Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, U.K.

Syntax:

$$\begin{array}{rcll} L, M, N & ::= & x & \text{variable} \\ & & M\,N & \text{application} \\ & & \lambda x.\, M & \text{abstraction} \end{array}$$

Free variables:

$$\begin{array}{rcl} FV(x) & = & \{x\} \\ FV(M\,N) & = & FV(M) \cup FV(N) \\ FV(\lambda x.\, M) & = & FV(M) - \{x\} \end{array}$$

Substitution:

$$\begin{array}{rcll} [N/x]x & = & N & \\ [N/x]z & = & z & \text{if } z \neq x \\ [N/x](L\,M) & = & ([N/x]L)\ ([N/x]M) & \\ [N/x](\lambda z.\, M) & = & \lambda z.\, ([N/x]M) & \text{if } z \neq x \text{ and } z \notin FV(N) \end{array}$$

Renaming of bound variables:

$$\lambda x.\, M = \lambda y.\, ([y/x]M) \quad \text{if } y \notin FV(M)$$

Operational Semantics:

$$(\lambda x.\, M)\, N \longrightarrow [N/x]M \quad \text{function application (“beta-reduction”)}$$

Figure 1: Syntax and Operational Semantics of the Lambda-Calculus

where

$$\mathit{Factorial}(n) \quad = \quad \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n \cdot \mathit{Factorial}(n-1).$$

For each nonnegative number n, instantiating the function *Factorial* with the argument n yields a number, the factorial of n, as result. Writing “$\lambda n.$” as a shorthand for “the function that, for each n, yields...,” we can restate the definition of *Factorial* as

$$\mathit{Factorial} \quad = \quad \lambda n.\ \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ n \cdot \mathit{Factorial}(n-1).$$

The expression $\mathit{Factorial}(0)$ is now read as “the function ‘$\lambda n.\ \mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ \mathit{etc.}$’ applied to the argument 0,” i.e., “the value that results when the bound variable n in the function body ‘$\mathit{if}\ n = 0\ \mathit{then}\ 1\ \mathit{else}\ \mathit{etc.}$’ is replaced by 0,” i.e. “$\mathit{if}\ 0 = 0\ \mathit{then}\ 1\ \mathit{else}\ \mathit{etc.}$,” i.e. 1.

In the 1930s, Church invented a mathematical system called the **lambda-calculus** (or λ-calculus) that embodies this kind of function definition and application in a pure form. In the lambda-calculus *everything* is a function: the arguments accepted by functions are themselves functions and the result returned by a function is another function.

2.1 Syntax and Operational Semantics

The syntax of the lambda-calculus comprises the three forms of expression at the top of Figure 1. A variable x by itself is a lambda-expression; the application of a lambda-expression M to another lambda-expression N, written $M\,N$, is a lambda-expression; and the abstraction of a variable x from a lambda-expression M, written $\lambda x.\, M$, is a lambda-expression. L, M, and N are used throughout this chapter to stand for arbitrary lambda-expressions. To avoid writing too many parentheses, application is taken to be left-associative, so that $L\,M\,N$ is the same as $(L\,M)\,N$, and the bodies of abstractions extend as far to the right as possible, so that $\lambda x.\,\lambda y.\,x\,y\,x$ is the same as $\lambda x.\,(\lambda y.\,((x\,y)\,x))$. The variable x is said to be **bound** in the body M of $\lambda x.\,M$.

In its pure form, the lambda-calculus has no built-in constants or operators — no numbers, arithmetic operations, records, loops, sequencing, I/O, etc. The sole means by which expressions "compute" is the application of functions to arguments, which is captured formally by the rule at the bottom of Figure 1, traditionally called **beta-reduction**. This rule says that an expression can be **reduced** by replacing some subexpression of the form $(\lambda x.\,M)N$, called a **redex**, by the result of substituting the argument N for the bound variable x in the body M. For example, $(\lambda x.\,x\,y)\,(u\,v)$ reduces to $u\,v\,y$, while $\underline{(\lambda x.\,\lambda y.\,x)\,z}\,w$ reduces (by the underlined redex) to $(\lambda y.\,z)\,w$, which further reduces to z. We write $(\lambda x.\,\lambda y.\,x)\,z\,w \longrightarrow^* z$ to show that the first expression reduces to the second by some sequence of steps of reduction.

The notion of reduction gives rise to a natural definition of what it means for two expressions to be "the same modulo reduction." M and N are **beta-convertible**, written $M =_\beta N$, if they are identical, or if one reduces to the other, or if they are each convertible to some third expression L. (Formally, this is summarized by saying that the beta-conversion relation is the reflexive, symmetric, transitive closure of beta-reduction.) For example, $(\lambda x.\,x)\,z$ and $(\lambda x.\,\lambda y.\,x)\,z\,w$ are convertible because they both reduce to z.

To make the notions of beta-reduction and conversion completely precise, there is a little technical work to be done. In particular, we must define the **substitution** notation $[N/x]M$. The details, which can be skipped on a first reading, occupy the rest of Figure 1 and of this subsection.

First, we say what it means for a variable x to be **free** in an expression M: namely, that x appears at some position in M where it is not bound by an enclosing lambda-abstraction on x. (For example, x is free in $x\,y$ and $\lambda y.\,x\,y$, but not in $\lambda x.\,x$ or $\lambda z.\,\lambda x.\,\lambda y.\,x\,y\,z$.)

An important syntactic convention concerns the inessentiality of bound names. Intuitively, it is clear that the expressions $\lambda x.\,x$ and $\lambda y.\,y$ describe exactly the same function — the function that, given any argument N, returns N. The fact that we use x in one case and y in the other to stand for the argument in the body is of no consequence. This principle is captured by the rule of **renaming of bound variables** (often called **alpha-conversion**) on the second line from the bottom of Figure 1, which states that we may freely replace the bound variable x by another variable y in a lambda-abstraction $\lambda x.\,M$ as long as y is not among the free variables of M. (The side condition is needed because, for example, we do not want to consider $\lambda x.\,y$ and $\lambda y.\,y$ to be the same function.)

Now we define substitution. Substituting an expression N for a variable x in an expression consisting only of x itself yields N. Substituting N for x in an expression consisting only of a different variable z yields z. To substitute N for x in an application $M\,L$, we substitute in M and L separately. To substitute N for x in a lambda-abstraction $\lambda z.\,M$, we substitute in the body M; however, we do this only when z is not the same as x and z is not one of the free variables of N. The first part of this side condition ensures that we do not allow nonsensical reductions like $(\lambda x.\,\lambda x.\,x)\,y \longrightarrow \lambda x.\,y$, where x is replaced by y even though it is actually bound by an inner abstraction, not the one being reduced. The second prevents a similar kind of mistake where free variables of N are "captured" by abstractions inside a term being substituted into, resulting in reductions like $(\lambda x.\,\lambda y.\,x)\,y \longrightarrow \lambda y.\,y$.

Thus, strictly speaking, the substitution $[N/x]M$ is undefined for some values of N, x, and M. But it is always possible to change the names of bound variables in N and/or M so that the substitution makes sense. For example, we can rewrite $[y/x](\lambda x.\, x)$ as $[y/x](\lambda z.\, z)$, which, by the definition of substitution, equals $\lambda z.\, z$; after renaming the bound variable again, this is the same as $\lambda x.\, x$. It is common practice to elide the renaming steps and say that $[y/x](\lambda x.\, x) = \lambda x.\, x$.

2.2 Examples

The lambda-calculus is much more powerful than its tiny definition might suggest. For example, there is no built-in provision for multi-argument functions, but it is easy to achieve the same effect using **higher-order functions** that yield functions as results. Suppose that M is an expression involving two free variables x and y and we want to write a function F that, for each pair (N, L) of arguments, yields the result of substituting N for x and L for y in M. Instead of writing $F = \lambda(x, y).\, M$, as we might in a higher-level programming language, we write $F = \lambda x.\, \lambda y.\, M$; that is, F is a function that, given a value N for x, yields a function that, given a value L for y, yields the desired result. We then apply F to its arguments one at a time, writing $F\, N\, L$, which reduces to $(\lambda y.\, [N/x]M)\, L$ and then to $[L/y][N/x]M$. This transformation of multi-argument functions into higher-order functions is often called **Currying** after its popularizer, Curry. (It was actually invented by Schönfinkel, but the term "Schönfinkeling" has not caught on.)

Another common language feature that can easily be encoded in the lambda-calculus is boolean values and conditionals. Define the lambda-expressions *True* and *False* as follows:

$$\begin{array}{rcl} \mathit{True} & = & \lambda t.\, \lambda f.\, t \\ \mathit{False} & = & \lambda t.\, \lambda f.\, f \end{array}$$

Both of these expressions are **combinators**; that is, neither contains any free variables. This means that they are inert with respect to substitution: $[N/x]\mathit{True} = \mathit{True}$ no matter what N and x are. The only way to "interact" with combinators is by applying them to other expressions. For example, we can use application to define a combinator *If* with the property that $\mathit{If}\ L\ M\ N$ reduces to M when $L = \mathit{True}$ and reduces to N when $L = \mathit{False}$.

$$\mathit{If} \quad = \quad \lambda l.\, \lambda m.\, \lambda n.\ l\ m\ n$$

The *If* combinator does not actually do much: $\mathit{If}\ L\ M\ N$ means just $L\ M\ N$. In effect, the boolean value L itself is the conditional: it takes two arguments and chooses the first (if it is *True*) or the second (if it is *False*). For example, the expression $\mathit{If}\ \mathit{True}\ M\ N$ reduces as follows:

$$\begin{array}{rcll} \mathit{If}\ \mathit{True}\ M\ N & = & \underline{(\lambda l.\, \lambda m.\, \lambda n.\ l\ m\ n)\ \mathit{True}}\ M\ N & \text{by definition} \\ & \longrightarrow & \underline{(\lambda m.\, \lambda n.\ \mathit{True}\ m\ n)\ M}\ N & \text{reducing the underlined redex} \\ & \longrightarrow & \underline{(\lambda n.\ \mathit{True}\ M\ n)\ N} & \text{reducing the underlined redex} \\ & \longrightarrow & \mathit{True}\ M\ N & \text{reducing the underlined redex} \\ & = & \underline{(\lambda t.\, \lambda f.\, t)\ M}\ N & \text{by definition} \\ & \longrightarrow & \underline{(\lambda f.\, M)\ N} & \text{reducing the underlined redex} \\ & \longrightarrow & M & \text{reducing the underlined redex} \end{array}$$

(Strictly speaking, this calculation is only valid if the variables n, t, and f are not free in M or N; otherwise, some extra renaming steps are required.)

We can also write boolean operators like logical-and as functions:

$$\mathit{And} \quad = \quad \lambda b.\, \lambda c.\, b\, c\, \mathit{False}$$

That is, *And* is a function that, given two boolean arguments b and c, returns either c (if b is *True*) or *False* (if b is *False*); thus $\mathit{And}\ b\ c$ yields *True* if and only if both b and c are *True*.

Using booleans, we can encode pairs of values as lambda-expressions. Define:

$$\begin{array}{lcl} \mathit{Pair} & = & \lambda f.\, \lambda s.\, \lambda b.\, b\, f\, s \\ \mathit{Fst} & = & \lambda p.\, p\ \mathit{True} \\ \mathit{Snd} & = & \lambda p.\, p\ \mathit{False} \end{array}$$

That is, $\mathit{Pair}\ M\ N$ is a function that, when applied to a boolean b, applies b to M and N. By the definition of booleans, this application yields M if b is *True* and N if b is *False*, so the first and second projection functions *Fst* and *Snd* can be implemented simply by supplying the appropriate boolean. To check that $\mathit{Fst}\ (\mathit{Pair}\ M\ N) =_\beta M$, calculate as follows:

$$\begin{array}{lcll} \mathit{Fst}\ (\mathit{Pair}\ M\ N) & = & \mathit{Fst}\ (\underline{(\lambda f.\, \lambda s.\, \lambda b.\, b\, f\, s)\ M}\ N) & \text{by definition} \\ & \longrightarrow & \mathit{Fst}\ (\underline{(\lambda s.\, \lambda b.\, b\, M\, s)\ N}) & \text{reducing the underlined redex} \\ & \longrightarrow & \mathit{Fst}\ (\lambda b.\, b\, M\, N) & \text{reducing the underlined redex} \\ & = & \underline{(\lambda p.\, p\ \mathit{True})\ (\lambda b.\, b\, M\, N)} & \text{by definition} \\ & \longrightarrow & \underline{(\lambda b.\, b\, M\, N)\ \mathit{True}} & \text{reducing the underlined redex} \\ & \longrightarrow & \mathit{True}\ M\ N & \text{reducing the underlined redex} \\ & \longrightarrow^* & M & \text{as above} \end{array}$$

The encoding of numbers as lambda-expressions is only slightly more intricate. Define the **Church Numerals** C_0, C_1, C_2, etc., as follows:

$$\begin{array}{lcl} C_0 & = & \lambda z.\, \lambda s.\, z \\ C_1 & = & \lambda z.\, \lambda s.\, s\, z \\ C_2 & = & \lambda z.\, \lambda s.\, s\, (s\, z) \\ & \vdots & \\ C_n & = & \lambda z.\, \lambda s.\, \underbrace{s\, (s\, (\ldots (s}_{n \text{ times}}\, z)) \ldots) \end{array}$$

That is, each number n is represented by a combinator C_n that takes two arguments, z and s ("zero" and "successor"), and applies n copies of s to z. As with booleans and pairs, this encoding makes numbers into active entities: the number n is represented by a function that does something n times — a kind of active unary numeral.

We can define some common arithmetic operations on Church numerals as follows:

$$\begin{array}{lcl} \mathit{Plus} & = & \lambda m.\, \lambda n.\, \lambda z.\, \lambda s.\, m\, (n\, z\, s)\, s \\ \mathit{Times} & = & \lambda m.\, \lambda n.\, m\, C_0\, (\mathit{Plus}\ n) \end{array}$$

Here, *Plus* is a combinator that takes two Church numerals, m and n, as arguments, and yields another Church numeral — i.e., a function that accepts arguments z and s, applies s iterated n times to z (by passing s and z as arguments to n), and then applies s iterated m more times to the result. It is an instructive exercise to check, for example, that $\mathit{Plus}\ C_2\ C_1 =_\beta C_3$. The definition of *Times* uses another trick: since *Plus* takes its arguments one at a time, applying it to just one argument n yields the function that adds n to whatever argument it is given. Passing this function as the second argument to m and zero as the first argument means "apply the function that adds n to its argument, iterated m times, to zero," i.e., "add together m copies of n."

To test whether a Church numeral is zero, we must give it a pair of arguments Z and S such that applying S to Z one or more times yields *False*, while not applying it at all yields *True*. Clearly, we can take Z to be just *True*. As for S, we use a function that throws away its argument and always returns *False*.

$$\mathit{IsZero} \quad = \quad \lambda m.\ m\ \mathit{True}\ (\lambda x.\ \mathit{False})$$

Surprisingly, it is quite a bit more difficult to subtract using Church numerals. It can be done using the following rather impenetrable "predecessor function," which, given C_0 as argument, returns C_0 and, given C_{i+1}, returns C_i:

$$\begin{array}{rclrcl} \mathit{Pred} & = & \lambda m.\ \mathit{Fst}\ (m\ Z\ S) & & & \\ & & \text{where} & Z & = & \mathit{Pair}\ C_0\ C_0 \\ & & & S & = & \lambda p.\ \mathit{Pair}\ (\mathit{Snd}\ p)\ (\mathit{Plus}\ (\mathit{Snd}\ p)\ C_1) \end{array}$$

This definition works by using m as a function to apply m copies of the function S to the starting value Z. Each copy of S takes a pair of numerals $(\mathit{Pair}\ C_i\ C_j)$ as its argument and yields $(\mathit{Pair}\ C_j\ C_{j+1})$ as its result. So applying S m times to $(\mathit{Pair}\ C_0\ C_0)$ yields $(\mathit{Pair}\ C_0\ C_0)$ if $m = 0$ and $(\mathit{Pair}\ C_{m-1}\ C_m)$ otherwise. In both cases, the predecessor of m is found in the first component.

Other common datatypes like lists, trees, arrays, and variant records can be encoded using similar techniques. Of course, in most programming languages based on the lambda-calculus, such basic data types are added as primitive constants, rather than being encoded.

A lambda-expression containing no redexes is said to be in **normal form**. The lambda-expressions we have seen so far have all shared the property that, independent of the order in which redexes are chosen for reduction, eventually all the redexes are used up and a normal form is reached. But not all lambda-expressions have this property. For example, the **divergent** combinator

$$\Omega \quad = \quad (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$$

can never be reduced to a normal form. It contains just one redex, and reducing this redex yields exactly Ω again!

A similar but more useful example is the so-called Y combinator, which can be used to define recursive functions such as *Factorial*.

$$Y \quad = \quad \lambda f.\ (\lambda x.\, f\, (x\, x))\, (\lambda x.\, f\, (x\, x))$$

The crucial property of Y is that $Y\ F =_\beta F\ (Y\ F)$ for any F, as can be seen by the following calculation:

$$\begin{array}{rcl} Y\ F & = & \underline{(\lambda f.\ (\lambda x.\, f\, (x\, x))\, (\lambda x.\, f\, (x\, x)))\ F} \\ & \longrightarrow & \underline{(\lambda x.\, F\, (x\, x))\, (\lambda x.\, F\, (x\, x))} \\ & \longrightarrow & F\, ((\lambda x.\, F\, (x\, x))\, (\lambda x.\, F\, (x\, x))) \\ & \longleftarrow & F\, (\underline{(\lambda f.\ (\lambda x.\, f\, (x\, x))\, (\lambda x.\, f\, (x\, x)))\ F}) \\ & = & F\, (Y\, F) \end{array}$$

(Note that it is not quite the case that $Y\, F \longrightarrow^* F\, (Y\, F)$: the last arrow goes in the wrong direction. There is a slightly more complicated variant that does have this property.)

Now, suppose we want to write a recursive function definition of the form $F = \langle \mathit{body\ containing}\ F \rangle$ — i.e., we want to write a definition where the expression on the right-hand side of the equals uses the very function that we are defining, as in the definition of *Factorial*.

The intention is that the recursive definition should be "unrolled" at the point where it occurs; for example, the definition of *Factorial* would intuitively be written

$$\begin{array}{l} \mathit{if}\ n = 0\ \mathit{then}\ 1 \\ \mathit{else}\ n \cdot (\mathit{if}\ n-1 = 0\ \mathit{then}\ 1 \\ \qquad\qquad \mathit{else}\ n \cdot (\mathit{if}\ n-2 = 0\ \mathit{then}\ 1 \\ \qquad\qquad\qquad\qquad \mathit{else}\ (n-2) \cdot \ldots)) \end{array}$$

This effect can be achieved by defining $G = \lambda f.\ \langle \mathit{body\ containing}\ f \rangle$ and $F = Y\ G$, since then

$$\begin{array}{rcl} F & = & Y\,G \\ & =_\beta & G\,(Y\,G) \\ & =_\beta & \langle \mathit{body\ containing}\ (Y\,G) \rangle \\ & =_\beta & \langle \mathit{body\ containing}\ \langle \mathit{body\ containing}\ (Y\,G) \rangle \rangle \\ & & \mathit{etc.} \end{array}$$

For example, if we define the factorial function by

$$\begin{array}{rcl} \mathit{Fact} & = & \lambda \mathit{fact}.\,\lambda n.\ If\ (\mathit{IsZero}\ n)\ C_1\ (\mathit{Times}\ n\ (\mathit{fact}\ (\mathit{Pred}\ n))) \\ \mathit{Factorial} & = & Y\ \mathit{Fact}, \end{array}$$

then applying *Factorial* to the Church numeral for 2 leads to the following calculation:

$$\begin{array}{rcl} \mathit{Factorial}\ C_2 & = & Y\ \mathit{Fact}\ C_2 \\ & =_\beta & \mathit{Fact}\ (Y\ \mathit{Fact})\ C_2 \\ & =_\beta & (\lambda \mathit{fact}.\,\lambda n.\ \mathit{If}\ (\mathit{IsZero}\ n)\ C_1\ (\mathit{Times}\ n\ (\mathit{fact}\ (\mathit{Pred}\ n))))\ (Y\ \mathit{Fact})\ C_2 \\ & =_\beta & (\lambda n.\ \mathit{If}\ (\mathit{IsZero}\ n)\ C_1\ (\mathit{Times}\ n\ (Y\ \mathit{Fact}\ (\mathit{Pred}\ n))))\ C_2 \\ & =_\beta & \mathit{If}\ (\mathit{IsZero}\ C_2)\ C_1\ (\mathit{Times}\ C_2\ (Y\ \mathit{Fact}\ (\mathit{Pred}\ C_2))) \\ & =_\beta & \mathit{If}\ \mathit{False}\ C_1\ (\mathit{Times}\ C_2\ (Y\ \mathit{Fact}\ C_1)) \\ & =_\beta & \mathit{Times}\ C_2\ (Y\ \mathit{Fact}\ C_1) \\ & = & \mathit{Times}\ C_2\ (\mathit{Factorial}\ C_1) \end{array}$$

That is, $\mathit{Factorial}(2) = 2 \cdot \mathit{Factorial}(1)$. This calculation justifies our informal assertion that *Factorial* really implements the factorial function: it can easily be proved that $\mathit{Factorial}(C_n) =_\beta C_{!n}$ for each nonnegative number n, where $!n$ is the "real" factorial of n.

Here are some other well-known combinators:

$$\begin{array}{rcl} I & = & \lambda x.\,x \\ K & = & \lambda x.\,\lambda y.\,x \\ S & = & \lambda x.\,\lambda y.\,\lambda z.\ (x\ z)\ (y\ z) \end{array}$$

I is called the "identity combinator" because $I\,M =_\beta M$ for any M. K is a combinator that takes two arguments, throws away the second, and returns the first. (We called the same expression *True* before, but we are not thinking of it here as representing a boolean value.) S "distributes" its third argument to both its first and its second arguments. An interesting property of these three is that, between them, they contain all the power of lambda-abstraction, in the sense that for any combinator M, there is a combinator N such that $N =_\beta M$ and N can be written using just S, K, I, and application, with no variables or abstractions except those occuring in the definitions of S, K, and I. For example, $\mathit{False} =_\beta K\,I$.

2.3 Properties of Reduction

The importance of lambda-calculus in computer science comes from the fact that it is simultaneously powerful enough to form a realistic core for many programming languages and simple enough that its properties can be studied mathematically. Indeed, the study of the lambda-calculus now constitutes a branch of mathematics in its own right. We review a few classical results.

A lambda-expression containing more than one redex can be reduced in more than one way, leading, in general, to different results. For example, $K\ I\ (K\ \mathit{True}\ \mathit{False})$ reduces in one step to either I or $K\ I\ \mathit{True}$. Here, we can further reduce the second result, yielding I again; but we might worry that there could be some M such that M could be reduced to either N_1 or N_2 in such a way that N_1 and N_2 could never be "brought back together" by reducing them further. This situation would render the notation $N_1 =_\beta N_2$ nonsensical, since it would be hard to argue that N_1 and N_2 are equal in any behavioral sense. Fortunately, the following theorem, sometimes called the fundamental syntactic property of the lambda-calculus, guarantees that this cannot actually happen — i.e., that reduction in the lambda-calculus is **confluent**.

Theorem [Church-Rosser]: If M reduces to two different expressions, N_1 and N_2, then these further reduce to some common expression L. (In symbols: if $M \longrightarrow^* N_1$ and $M \longrightarrow^* N_2$, then there is some L such that $N_1 \longrightarrow^* L$ and $N_2 \longrightarrow^* L$.)

Corollary: If two terms are beta-convertible, then they both reduce to some common term — i.e., if $M =_\beta N$ then there is some L such that that $M \longrightarrow^* L$ and $N \longrightarrow^* L$.

The latter form of the Church-Rosser property immediately implies the **uniqueness of normal forms**: if $N_1 =_\beta N_2$ and neither N_1 nor N_2 contain any redexes, then N_1 and N_2 must be identical (modulo names of bound variables). This justifies regarding the normal form of a term (when it has one) as its "meaning" or "value." For example, we can say that the value of the expression $\mathit{Plus}\ C_3\ C_2$ is C_5.

We have seen that there are some terms, like Ω, that do not have normal forms. There are also some, like $K\,I\,\Omega$, that do have a normal form (I), but that can also be reduced indefinitely: $K\,I\,\underline{\Omega} \longrightarrow K\,I\,\underline{\Omega} \longrightarrow K\,I\,\underline{\Omega} \longrightarrow \mathit{etc.}$. We call terms like Ω **non-normalizable**, terms like $K\,I\,\Omega$ **normalizable**, and terms like $\mathit{Plus}\ C_1\ (\mathit{Plus}\ C_2\ C_3)$, for which *every* sequence of reductions ends in a normal form, **strongly normalizing**.

A **reduction strategy** is a rule specifying which redexes should be reduced first. There are several common reduction strategies (a classic comparison is [Plotkin, 1975]). The **normal-order** reduction strategy, sometimes called **call-by-name** reduction, always reduces the redex whose λ appears the furthest to the left. The **lazy** strategy also reduces the leftmost redex, but only if that redex is not itself in the body of some abstraction — that is, lazy reduction stops when the expression reaches a **weak head normal form** with no "top-level" redexes. Similarly, **applicative-order** (or **call-by-value**) reduction always chooses the leftmost redex $(\lambda x.\,M)\,N$ where N is in a particular "evaluated form": either a variable, or a lambda-abstraction, or a variable applied to some expression in evaluated form (in particular, *not* a redex). These strategies lead to different choices of which redex to reduce first in the following expression:

$\lambda v.\ \underline{(\lambda z.\,z)\,((\lambda w.\,w)\,(x\,(\lambda y.\,y)))}$	normal order
$\lambda v.\ (\lambda z.\,z)\,\underline{((\lambda w.\,w)\,(x\,(\lambda y.\,y)))}$	applicative order
$\lambda v.\ (\lambda z.\,z)\,((\lambda w.\,w)\,(x\,(\lambda y.\,y)))$	lazy (no reductions allowed)

Applicative-order reduction exactly matches the ordering of function calls in call-by-value programming languages such as Scheme and ML, while lazy reduction approximates the ordering in languages such as Haskell and Algol-60 [cf. Goldberg's article in this handbook]. Normal-order and lazy reduction are "safer" strategies than applicative-order, since they never fail to terminate except on non-normalizable expressions. For example, the expression $K\,I\,\Omega$ leads to the normal form I under normal-order reduction, while applicative-order leads to an infinite sequence of reductions of Ω.

Theorem [Normalization]: A sequence of normal-order reductions beginning from a normalizable term M always terminates in a normal form after a finite number of steps. Similarly, lazy reduction always finds a weak head normal form for any term that has one.

2.4 Operational and Denotational Equivalences

When should we say that two lambda-expressions "behave the same"? Two essentially different sorts of answer can be given.

On one hand, we can take an **operational** view, concentrating on how expressions behave under reduction. We have already seen one notion of equivalence that arises in this way, where two expressions are judged equivalent if they are beta-convertible. But we might reasonably wish to extend this notion a little, since there are pairs of expressions like $(\lambda x.\, x\,x)(\lambda x.\, x\,x)$ and $(\lambda x.\, x\,x\,x)(\lambda x.\, x\,x)$ that are not beta-convertible, but that nevertheless have arguably the same external behavior (i.e., none at all; just an infinite sequence of internal steps). On the other hand, simply saying "two terms are equivalent if they are convertible or if neither has a normal form" goes too far in the other direction, since there are some terms without normal forms that do "behave differently." For example, the expressions $\lambda x.\, x\; \mathit{True}\,\Omega$ and $\lambda x.\, x\; \mathit{False}\,\Omega$ are both non-normalizable, but applying the first to K yields an expression with normal form *True*, while applying the second to K yields an expression with normal form *False*. In other words, $\lambda x.\, x\; \mathit{True}\,\Omega$ and $\lambda x.\, x\; \mathit{False}\,\Omega$ have the same behavior in isolation (both diverge), but they do not have the same behavior in all contexts.

The idea that equivalent expressions should have the same behavior in all contexts can be formalized using the following notion of **contextual equivalence**, first studied by Morris. First, we introduce the notion of a **context**: a lambda-expression with a "hole," written $[\,]$, into which another expression can be placed. For example, suppose $C[\,]$ is the context $\lambda x.\, [\,]\, x$. Filling the hole in $C[\,]$ with the expression $x\,y$, written $C[x\,y]$, yields $\lambda x.\, x\,y\,x$. (Notice how, unlike substitution, the variable x in the expression $x\,y$ is "captured" by the binder λx.) Two expressions M and N are said to be equivalent when, for any context $C[\,]$, the expression $C[M]$ is normalizable if and only if $C[N]$ is normalizable.

Note that this definition is stated in terms of normalizability, and does not explicitly demand that the normal forms of M and N be the same. This notion of "observation" might initially seem too weak. For example, the context $C[\,] = [\,]\, K$ fails to distinguish the expressions $\lambda x.\, x\; \mathit{True}\,\Omega$ and $\lambda x.\, x\; \mathit{False}\,\Omega$, since $C[\lambda x.\, x\; \mathit{True}\,\Omega] \longrightarrow \mathit{True}$ and $C[\lambda x.\, x\; \mathit{False}\,\Omega] \longrightarrow \mathit{False}$ both reduce to normal forms. The discriminating power of contextual equivalence comes from the quantification over all contexts. Here, the more complex context $C[\,] = [\,]\, K\,\Omega\, I$ does distinguish the two processes in question, since $C[\lambda x.\, x\; \mathit{True}\,\Omega] \longrightarrow^* K\; \mathit{True}\,\Omega\,\Omega\, I \longrightarrow^* \mathit{True}\,\Omega\, I \longrightarrow^* \Omega$ has no normal form, while $C[\lambda x.\, x\; \mathit{False}\,\Omega] \longrightarrow^* K\; \mathit{False}\,\Omega\,\Omega\, I \longrightarrow^* \mathit{False}\,\Omega\, I \longrightarrow^* I$ has the normal form I.

An equivalent way of formulating the intuition behind contextual equivalence is via the notion of **applicative bisimulation**. The idea here is to give a method of testing whether two expressions have observably *different* behavior, and regard two expressions as equivalent if they cannot be shown to be different by making any finite sequence of tests. Formally, two combinators M and N are said to be bisimilar only if (1) either both are non-normalizable or both are normalizable, and (2) for each combinator L, the applications $M\,L$ and $N\,L$ are bisimilar. Two arbitrary expressions M and N, possibly with free variables, are bisimilar if, no matter what combinators are substituted for their free variables, the resulting combinators are bisimilar in the previous sense.

The form of the definition of applicative bisimulation, which takes all expressions to be bisimilar except those that fail one of the two conditions above, leads to a powerful "coinductive" reasoning technique: to show that two expressions are bisimilar, it suffices to show that no contradiction results from the assumption that they are. For example, to show that $\Omega = (\lambda x.\, x\, x)(\lambda x.\, x\, x)$ is bisimilar to $\Omega_1 = (\lambda x.\, x\, x\, x)(\lambda x.\, x\, x)$, we reason as follows: If they are *not* bisimilar, then there must be some sequence of terms $L_1,\ L_2,\ \ldots,\ L_k$ such that $\Omega\, L_1\, L_2\, \ldots\, L_k$ is normalizable and $\Omega_1\, L_1\, L_2\, \ldots\, L_k$ is not, or vice versa. But this cannot be, since both Ω and Ω_1 can only reduce to non-normalizable terms, no matter what they are applied to. So Ω and Ω_1 are bisimilar.

As this example illustrates, applicative bisimulation is typically much easier to use than contextual equivalence. In order to show directly that Ω and Ω_1 are contextually equivalent, we would have to show that they have the same behavior when placed in an *arbitrary* context; applicative bisimulation allows us to consider just contexts of the form $[]\, L_1\, L_2\, \ldots$

Many variants of the definitions of contextual equivalence and applicative bisimulation have been studied. For example, "normalizable" can be replaced by "reaches a normal form under a normal-order reduction strategy," or "... under a lazy strategy," etc.

A somewhat different, **denotational** perspective on expression equivalence is obtained by returning to the original intuition that lambda-expressions were intended to represent functions, and say that M and N are the same if they represent the same function — i.e., if they map the same inputs to the same outputs. To make this precise, we need to choose some **semantic domain** D (i.e., some set of functions) and define a **denotation function** $[\![\,]\!]$ that maps each expression M into an element $[\![M]\!]$ of D. (See Schmidt's article in this handbook for more discussion of domains and denotation functions.) There are some serious technical problems with defining D and $[\![\,]\!]$, stemming from the fact that, since lambda-expressions take lambda-expressions as arguments and return lambda-expressions as results, each element of D is actually a function from D to D — that is, D must be a larger set than the set of functions from D to D. Trying to construct such a D naively leads to a mathematical paradox, where we end up with a D that is strictly larger than itself. Fortunately, Scott and Plotkin realized in 1969 that, by considering only some of the functions from D to D, the paradox can be avoided. Indeed, the same basic insight can be used to construct many different Ds with different properties.

2.5 Research Areas

The study of denotational semantic models of the lambda-calculus and related systems has led to a rich research literature, surveyed in textbooks by BarendregtBarendregt, Gunter [1992], Schmidt [1986], and Winskel [1993] and a shorter article by Gunter and Scott [Gunter and Scott, 1990]. One issue that has received considerable attention is the problem of finding "fully abstract" models, in which each lambda-expression is mapped to an element of the model (a mathematical function of some carefully chosen sort) in such a way that two lambda-expressions have equivalent operational behavior if and only if they denote the same element of the model.

Operational notions of program equivalence such as applicative bisimulation have begin to recieve serious attention only relatively recently (e.g. [Gordon, 1994]).

Implementation techniques for programming languages based on the lambda-calculus have a long history, from Landin's original "SECD machine" to more modern proposals such as the G-machine [Peyton Jones and Lester, 1992]. A related theoretical development is work on "optimal" reduction strategies, which try to choose redexes so as to reach a normal form as quickly as possible. The lambda-calculus forms a common basis for work on optimization techniques such as partial evaluation [Jones *et al.*, 1993]; related notations are being used as intermediate languages in optimizing compilers for high-level languages such as C.

In "impure" functional languages such as Scheme and ML, mutable variables are added to the λ-calculus, retaining the higher-order flavor of the pure calculus while giving up the simple intuition that expressions represent mathematical functions (cf. Goldberg's chapter in this handbook). For imperative computation, where evaluation of an expression can also have "side effects" on mutable variables, it is not yet clear how to reason about equivalence of expressions, since there is no obvious choice for the definition of what is observable. For example, mutable variables may be local to a particular function, and side-effects on these are not directly observable in the same way as side-effects on global variables, though they can be observed indirectly by observing the input-output behavior of the function.

One of the most active areas of lambda-calculus research is the study of typed lambda-calculi, in which functions are classified according to the types of arguments they can "correctly" accept and the types of results they can return. Such calculi typically have quite different properties from the untyped calculus presented here. For example, it is typically the case that every term is strongly normalizing (combinators like Ω and Y cannot then be defined). Also, semantic models of typed calculi are often more straightforward to construct. See Cardelli's chapter in this handbook.

3 Pi-Calculus

The lambda-calculus holds an enviable position: it is recognized as embodying, in miniature, all of the essential features of functional computation. Moreover, other foundations for functional computation, such as Turing machines, have exactly the same expressive power. The "inevitability" of the lambda-calculus arises from the fact that the only way to observe a functional computation is to watch which output values it yields when presented with different input values.

Unfortunately, the world of concurrent computation is not so orderly. Different notions of what can be observed may be appropriate in different circumstances, giving rise to different definitions of when two concurrent systems have "the same behavior": for example, we may wish to observe or ignore the degree of inherent parallelism of a system, the circumstances under which it can deadlock, the distribution of its processes among physical processors, or its resilience to various kinds of failures. Moreover, concurrent systems can be described in terms of many different constructs for creating processes (fork/wait, cobegin/coend, futures, data parallelism, etc.), exchanging information between them (shared memory, rendezvous, message-passing, dataflow, etc.), and managing their use of shared resources (semaphores, monitors, transactions, etc.).

This variability has given rise to a large class of formal systems called **process calculi** (sometimes **process algebras**), each embodying the essence of a particular concurrent or distributed programming paradigm. We focus here on one typical process calculus, the **pi-**

Syntax:

$$
\begin{array}{rcll}
P, Q, R & ::= & \mathbf{0} & \text{inert process} \\
& & x(y).\,P & \text{input prefix} \\
& & \bar{x}y.\,P & \text{output prefix} \\
& & P \mid Q & \text{parallel composition} \\
& & (\nu x)\,P & \text{restriction} \\
& & !P & \text{replication}
\end{array}
$$

Renaming of bound variables:

$$
\begin{array}{rcll}
x(y).\,P & = & x(z).([z/y]P) & \text{if } z \notin FV(P) \\
(\nu y)\,P & = & (\nu z)\,([z/y]P) & \text{if } z \notin FV(P)
\end{array}
$$

Structural Congruence:

$$
\begin{array}{rclll}
P \mid Q & \equiv & Q \mid P & & \text{commutativity of parallel composition} \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & & \text{associativity of parallel composition} \\
((\nu x)P) \mid Q & \equiv & (\nu x)\,(P \mid Q) & \text{if } x \notin FV(Q) & \text{``scope extrusion''} \\
!P & \equiv & P \mid !P & & \text{replication}
\end{array}
$$

Operational Semantics:

$$
\begin{array}{rclll}
\bar{x}y.P \mid x(z).Q & \longrightarrow & P \mid [y/z]Q & & \text{communication} \\
P \mid R & \longrightarrow & Q \mid R & \text{if } P \longrightarrow Q & \text{reduction under } \mid \\
(\nu x)P & \longrightarrow & (\nu x)Q & \text{if } P \longrightarrow Q & \text{reduction under } \nu \\
P & \longrightarrow & Q & \text{if } P \equiv P' \longrightarrow Q' \equiv Q & \text{structural congruence}
\end{array}
$$

Figure 2: Syntax and Operational Semantics of the Pi-Calculus

calculus (or π-calculus) of Milner, Parrow, and Walker [Milner *et al.*, 1992, Milner, 1991]. References to some other popular process calculi can be found at the end of the section.

In the pure lambda-calculus, everything is a function; numbers, for example, are encoded as special functions that can be interrogated (by applying them) to find out which number they represent. Analogously, in the pi-calculus, every expression denotes a **process** — a free-standing computational activity, running in parallel with other processes and possibly containing many independent subprocesses. Two processes can interact by exchanging a message on a **channel**. Indeed, communication along channels is the sole means of computation, just as function application is in the lambda-calculus. The only thing that can be observed about a process's behavior is its ability to send and receive messages.

3.1 Syntax and Operational Semantics

The simplest pi-calculus expression is the "inert process" $\mathbf{0}$, which denotes a process with no behavior at all. More interestingly, if P is some process expression, then the expression $x(y).\,P$ denotes a process that waits to read a value y from the channel x and then, having received it, behaves like P. Similarly, $\bar{x}y.\,P$ denotes a process that first waits to send the value y along the channel x and then, after y has been accepted by some input process, behaves like P.

$P \mid Q$ denotes a process composed of two subprocesses, P and Q, running in parallel. That is, $P \mid Q$ can exhibit all of the observable behaviors (sequences of messages sent and received) of both P and Q, interleaved in any order. Moreover, if P can send a message on some channel x and Q can receive on x, then $P \mid Q$ can perform an internal communication in which the message is exchanged between P and Q.

Placing the restriction operator (νx) before a process expression P ensures that x is a fresh channel in P — i.e., that messages sent and received by P on x will never be mixed with messages sent or received on any other channel created elsewhere, even another channel that happens to be named x. That is, the alphabetic names of channels are unimportant, just as the names of bound variables are unimportant in the lambda-calculus: the process $(\nu x)\, \bar{y}x.\, \mathbf{0}$ is completely equivalent to $(\nu z)\, \bar{y}z.\, \mathbf{0}$ — both introduce a fresh channel, different from all other channels, and send it on y.

Finally, the "replicated process" $!P$ stands for an infinite number of copies of P, all running in parallel. Typically, only a few copies will actually be doing anything at a given moment; $!P$ should really be thought of as a simple notational device for describing processes with infinitely *long* sequences of behaviors. For example, $!(t(w).\, \bar{x}y.\, \bar{t}v.\, \mathbf{0})$ denotes a process that, after it is "triggered" by someone sending it a message on t, sends a message on x and then triggers another copy of itself by sending another message on t — i.e., it responds to a message on t by sending an infinite stream of messages on x.

When an input or output prefix is followed by $\mathbf{0}$, we normally drop the $\mathbf{0}$, writing $\bar{x}y$ instead of $\bar{x}y.\, \mathbf{0}$. Also, to avoid writing too many parentheses, we give input and output prefixes the strongest precedence, replication the next strongest, parallel composition the next, and restriction the weakest, so that $(\nu x)\, !x(y).\, a(b) \mid \bar{z}w$ means $(\nu x)\, ((!(x(y).\, a(b).\, \mathbf{0})) \mid \bar{z}w.\, \mathbf{0})$.

The definitions of free variables and substitution in the pi-calculus are similar to the corresponding definitions in the lambda-calculus. The expressions $x(y).\, P$ and $(\nu x)\, P$ bind the variable x in the body P. As before, we silently rename bound variables whenever necessary. Substitution is simpler here, because we only substitute variables for variables — we do not substitute processes into other processes.

The operational semantics of pi-calculus expressions is defined as a **reduction** relation, as in the lambda-calculus. We say that P reduces to Q, written $P \longrightarrow Q$, if P contains two parallel subprocesses that can communicate on some channel x to become the corresponding subprocesses of Q. This is formalized by the group of rules at the bottom of Figure 2. The first rule, corresponding to the beta-reduction rule in lambda-calculus, defines a primitive step of communication: a redex consisting of an output process $\bar{x}y.\, P$ in parallel with an input process $x(z).\, Q$ reduces to P in parallel with Q, where the data value y is substituted for the bound variable z in Q. For example, $\bar{x}y.\, \bar{y}z \mid x(w).\, w(v)$ reduces to $\bar{y}z \mid y(v)$, which further reduces to $\mathbf{0} \mid \mathbf{0}$, which cannot reduce further. This example underscores the fact that the "data value" that is passed from sender to receiver during a communication is itself a channel, and may later be used by the receiver for communication.

The next two rules specify that, if a communication can occur between two subprocess of a process P, then the same communication can still occur when P is placed in parallel with another process Q and, similarly, if a communication can occur within P, then the same communication can occur within $(\nu x)\, P$. Note that we do not allow reductions to occur inside the body of a process that is prefixed by an input or output.

The final rule in Figure 2 allows the two "halves" of a redex to be mixed together with other parallel components of a larger system. For example, in the expression $x(y).\, P \mid \bar{x}y.\, Q \mid \bar{x}z.\, R$, there are two possible communications on x (between the first and second components and between the first and third), but neither has exactly the form of the left-hand side of the communication rule: the second redex has an extra subprocess "in the middle," and both redexes are in the wrong order, with the input subprocess first. The **structural congruence** relation $\equiv$ formalizes the intuition that this doesn't matter — that $x(y).P \mid \bar{x}y.Q \mid \bar{x}z.R$ is just a another way of writing $\bar{x}y.Q \mid x(y).P \mid \bar{x}z.R$ or $\bar{x}z.R \mid x(y).P \mid \bar{x}y.Q$, both of which literally contain redexes. The four rules defining $\equiv$ may be applied any number of times within a process expression. The structural congruence rule for reduction has the effect that P can reduce to Q whenever P can be rearranged so that it literally contains a redex by which it can reduce to Q. In addition to rearranging the order of parallel compositions — the task of the first two of the structural congruence rules — there are rules for rearranging ν and !. The first says that the scope of a ν binding may be enlarged to enable reduction, as in $((\nu z)\,\bar{x}z.P) \mid x(y).Q \equiv (\nu z)\,\bar{x}z.P \mid x(y).Q \longrightarrow (\nu z)\,(P \mid [z/y]Q)$. The second formalizes the intuition that $!P$ behaves just the same as an arbitrary number of parallel copies of P: we can move new copies of P out from under the ! at will, making them available to participate in communications.

Because of the structural congruence rule, each process expression beginning with an input or an output may, in general, be part of several redexes at once. For example, the expression $(\nu x)\,\bar{x}y \mid \bar{x}z \mid x(w).\bar{w}v$ contains the redexes $(\nu x)\,\underline{\bar{x}y} \mid \bar{x}z \mid \underline{x(w).\bar{w}v}$ and $(\nu x)\,\bar{x}y \mid \underline{\bar{x}z} \mid \underline{x(w).\bar{w}v}$, both of which include $x(w).\bar{w}v$. Reducing either one of these has the effect of destroying the other. Moreover, the resulting processes are irretrievably different: one reduction yields $(\nu x)\,\bar{x}z \mid \bar{y}v$, whose only further behavior is sending a message on y, while the other yields $(\nu x)\,\bar{x}y \mid \bar{z}v$, which can only send a message on z. This **non-confluence** of reduction in the pi-calculus is crucial, since it models the fact that real concurrent programs may often yield different results depending on the order in which various internal events occur. Such timing dependencies may be undesirable (they are often called "race conditions"), but we need a framework in which they *can* occur in order for the assertion that they do *not* occur in a particular process to have any force! A related point is that we are not interested in a conversion relation on processes, as we were in the lambda-calculus: the fact that P and Q can both reduce to R is not sufficient reason to claim that P and Q behave the same, since P and Q may also be able to reduce to other, completely dissimilar, processes.

3.2 Examples

As with the lambda-calculus, the simplicity of the pi-calculus is deceptive: the primitive mechanisms of restriction, communication, parallel composition, and replication can be used to model a great variety of programming structures.

One very easy encoding trick allows several values to be sent and received in each message on a channel. Write $\bar{x}\langle y_1, \ldots, y_n\rangle$ for the simultaneous output of the tuple y_1 through y_n on x, and $x(z_1, \ldots, z_n)$ for the corresponding input process, which accepts a tuple of values and binds them to the variables z_1 to z_n. The case where $n = 0$ corresponds to sending a data-less signal on x. These **polyadic** communication prefixes can be encoded in the basic pi-calculus as follows:

$$\begin{array}{rcll} \bar{x}\langle y_1, \ldots, y_n\rangle.P & = & (\nu p)\,\bar{x}p.\bar{p}y_1.\ldots\bar{p}y_n.P & \qquad \text{choosing } p \notin FV(P) \\ x(z_1, \ldots, z_n).Q & = & x(p).p(z_1).\ldots p(z_n).Q & \qquad \text{choosing } p \notin FV(Q) \end{array}$$

That is, to send y_1 through y_n on x, we make up a private channel p, send p on x, and then send y_1 through y_n, one after the other, on p. Conversely, to read a tuple from x, we read p from x and then read the values from p. By using use a fresh p each time we send a tuple on x, we avoid any possible confusion between different processes sending tuples on x at the same time.

The encodings of common data structures in the lambda-calculus all have their counterparts in the pi-calculus. For example, the boolean values *True* and *False* are encoded by processes that repeatedly accept (over some channel b) a pair of channels t and f and respond by sending a message on either t or f:

$$\begin{array}{lcl} \mathit{True}(b) & = & !b(t,f).\,\bar{t}\langle\rangle \\ \mathit{False}(b) & = & !b(t,f).\,\bar{f}\langle\rangle \end{array}$$

If $\mathit{Test}(b)$ is defined as $(\nu t)\,(\nu f)\,\bar{b}\langle t,f\rangle.\,(t().\,P \mid f().\,Q)$, then the composite process $\mathit{True}(b) \mid \mathit{Test}(b)$ reduces to $\mathit{True}(b) \mid P \mid (\nu t)\,(\nu f)\,(\mathbf{0} \mid f().\,Q)$, while $\mathit{False}(b) \mid \mathit{Test}(b)$ reduces to $\mathit{False}(b) \mid Q \mid (\nu t)\,(\nu f)\,(\mathbf{0} \mid t().\,P)$, where the third subprocess in each case is unable to participate in any further actions. (For notational convenience, some variants of the pi-calculus include extra structural congruence rules like $P \mid \mathbf{0} \equiv P$ to "garbage-collect" such deadlocked subprocesses.) The channel b can be thought of as the **location** of the value $\mathit{True}(b)$, since b is the only means by which other processes can refer to it or interact with it, or as a **reference** to (the process encoding) the value *True*.

The pi-calculus is a quintessentially imperative (i.e. non-functional) language, in the sense that we do not expect to get the same result each time if we query a process repeatedly over a channel. A typical example is a "reference cell object," which maintains a single piece of state, updating it in response to messages sent over a channel w and reporting its current value in response to messages sent over a channel r.

$$\begin{array}{lcl} \mathit{Ref}(r,w,i) & = & (\nu l)\ \bar{l}\langle i\rangle \mid \mathit{ReadServer}(l,r) \mid \mathit{WriteServer}(l,w) \\ \mathit{ReadServer}(l,r) & = & !r(c).\,l(v).\,(\bar{c}\langle v\rangle \mid \bar{l}\langle v\rangle) \\ \mathit{WriteServer}(l,w) & = & !w(c,v').\,l(v).\,(\bar{c}\langle\rangle \mid \bar{l}\langle v'\rangle) \end{array}$$

The process $\mathit{Ref}(r,w,i)$ comprises three parts: a process waiting to send the initial value i on an internal channel l, which is private to the reference cell; a "read server" that accepts messages on the channel r and, in response to each, sends back the current value on a result channel c that is included in the message; and a "write server" that accepts messages containing new values for the reference cell and acknowledges their acceptance on a completion channel c that is included in the message. Each time a read or write request is received, a process is created that reads the current value of the reference cell from the internal channel l, responds to the request by sending something back on the result or completion channel c, and then restores either the existing value or the specified new value of the cell by sending it on l. At any given moment, there will be just one process ready to send on l; this invariant ensures that, for example, the client process

$$(\nu c)\ \bar{w}\langle v,c\rangle.\,c().\,(\nu d)\,\bar{r}\langle d\rangle.\,d(e).\,Q$$

will always receive v on d in response to its request message on r, assuming that no other client processes are simultaneously interacting with the same reference cell (note that it waits for the write-completion signal to arrive on c before sending the read request).

As a final illustration of the power of the pi-calculus, here is an encoding of the lambda-calculus itself, with a normal-order reduction strategy. Given a lambda-expression M and a channel p, define the process expression $[\![M]\!](p)$ as follows:

$$\begin{array}{lcl} [\![\lambda x.\,M]\!](p) & = & p(x,q).[\![M]\!](q) \\ [\![x]\!](p) & = & \bar{x}\langle p\rangle \\ [\![M\,N]\!](p) & = & (\nu q)\,[\![M]\!](q) \mid ((\nu y)\ \bar{q}\langle y,p\rangle \mid !y(r).\,[\![N]\!](r)) \end{array}$$

$[\![M]\!](p)$, pronounced "the process representing M with argument port p," is an expression that rests dormant until it receives on p a "trigger" x for its argument and a new argument port q. It then evolves to a new process with argument port q. For example, the lambda-expression $(\lambda x.\, x)z$ is translated as follows:

$$\begin{array}{rcl}
[\![(\lambda x.\, x)z]\!](p) & = & (\nu q)\, [\![\lambda x.\, x]\!](q) \mid ((\nu y)\, \bar{q}\langle y, p\rangle \mid !y(r).\, [\![z]\!](r)) \\
 & = & (\nu q)\, (\underline{q(x, q').\, \bar{x}\langle q'\rangle}) \mid ((\nu y)\, \underline{\bar{q}\langle y, p\rangle} \mid !y(r).\, \bar{z}\langle r\rangle) \\
 & \longrightarrow & (\nu q)\, (\nu y)\, \underline{\bar{y}\langle p\rangle} \mid !\underline{y(r).\, \bar{z}\langle r\rangle} \\
 & \longrightarrow & (\nu q)\, (\nu y)\, (!y(r).\, \bar{z}\langle r\rangle) \mid \bar{z}\langle p\rangle \\
 & \text{"="} & \bar{z}\langle p\rangle \\
 & = & [\![z]\!](p)
\end{array}$$

3.3 Operational Equivalence

The "=" in the penultimate line above deserves some discussion. Informally, it is clear that the process $(\nu q)\, (\nu y)\, (!y(r).\, \bar{z}\langle r\rangle) \mid \bar{z}\langle p\rangle$ can only communicate on z, since the replicated input on y appears immediately underneath the binder (νy), which guarantees that there can never be any sender on y. How do we express rigorously such assertions that one process expression has the same behavior as another?

As with the lambda-calculus, there are two basic approaches: either we can try to identify some class of "real processes" and say that two process expressions are equivalent if they denote the same real process, or we can focus on the reduction relation of the process calculus itself and say that two process expressions are equivalent if they have the same reduction behavior in all contexts. Work is proceeding on both fronts, but the situation is much more complex than in the simple world of functional computation (cf. [Milner, 1990, Baeten and Weijland, 1990, Hennessey, 1988, Hoare, 1985] for discussion and references). For denotational approaches, the problem is finding a pragmatically satisfying definition of "real processes." For operational approaches, the (related) problem is that there are many possible definitions of "behavior," yielding numerous, subtly different, equivalences [van Glabbeek, 1993]. So far, operational techniques have proved most successful, but both remain active research areas.

As in lambda-calculus, the most intuitive way of defining operational equivalence is via some notion of **contextual equivalence**. A **process context** is a process expression with a hole into which another process can be placed. We say that P and Q are equivalent when $C[P]$ and $C[Q]$ have the same "observable behavior" for each process context $C[\,]$.

In the lambda-calculus, we saw that there were several variations on the definition of contextual equivalence, depending on precisely what **observations** we allow of a lambda term M in some testing context $C[\,]$. For example, we might choose to observe whether $C[M]$ is normalizable, whether it is normalizable using a normal-order reduction strategy, whether it is strongly normalizing, etc. — each choice giving rise to a different notion of equivalence of lambda-terms. In the pi-calculus, there are even more choices. If $C[\,]$ is a testing context for a process P, we might choose to observe, for example,

1. whether $C[P]$ can eventually perform an input or output action;

2. what *sequences* of input and output actions $C[P]$ can perform;

3. how early or late $C[P]$ becomes committed to producing certain sequences of inputs and outputs (so that, for example, we distinguish the case where $C[P]$ can send on a and then chooses whether to send on b or on c from the case where $C[P]$ chooses first whether to send on a and then b or on a and then c);

4. the circumstances under which $C[P]$ can become deadlocked.

Note that "does $C[P]$ have a normal form?" is *not* a useful kind of observation in the pi-calculus, since it leads us to regard all processes with infinite behaviors as behaviorally identical. In the lambda-calculus, this identification makes sense, since any lambda-expression whose reduction does not terminate can be viewed as an infinite loop. But in the pi-calculus, a process may go on computing forever but still interact usefully with its environment. Real parallel and distributed systems are full of server processes with this property.

As in the lambda-calculus, contextual equivalence between two processes can be difficult to establish, because it demands that they must yield the same observable behavior when placed in an arbitrary testing context $C[\,]$. Fortunately, some useful variants of contextual equivalence can reformulated in terms of direct conditions on the processes themselves, with no quantification over contexts. For example, the following definition coincides with an observational congruence where the allowed observations are those listed second and third above.

We say that two process expressions P and Q are **bisimilar** if every action of one can be matched by a corresponding action of the other to reach a bisimilar state. More precisely, (1) if P can output the value a on the channel x and become P', the Q must also be able to output a on x and reach Q' such that P' is bisimilar to Q'; (2) if P can input a value a from the channel x and become P', the Q must also be able to input a from x and reach Q' such that P' is bisimilar to Q'; (3) if P can perform some internal communication to become P', then Q must be able to perform an internal communication to reach a state Q' such that P' and Q' are bisimilar; and three similar clauses where the roles of P and Q are reversed. We are being informal here about what it means that a process can input or output a on x and become another process; this can be formulated precisely using **labelled transition systems**.

Like applicative bisimulation for lambda-terms, the definition of bisimulation for process expressions is given in a **coinductive** style: two processes are bisimilar if we cannot show that they are not — or, equivalently, if assuming that they are leads to no contradictions. This style of definition gives rise to a powerful technique for proving the bisimilarity of processes, reminiscent of familiar inductive proof techniques used to prove properties of recursive functions in functional programming languages. We illustrate this technique with a simple example.

Suppose we want to show that the process $(\nu q)\,(\nu y)\,(!y(r).\,R) \mid S$ is bisimilar to S, as long as R and S do not have q and y among their free variables. Begin by assuming that this is true. To check that there are no contradictions, we choose an arbitrary R and S and check that the three conditions in the definition of bisimulation are satisfied (in both directions). For the first condition, suppose $(\nu q)\,(\nu y)\,(!y(r).\,R) \mid S$ can make an output a on some channel x and become P'. Since the subexpression $!y(r).\,R$ begins with an input, it is clear that an output on x must come from S, so $P' = (\nu q)\,(\nu y)\,(!y(r).\,R) \mid S'$ for some S', and that S in isolation can make the same output and become S'. Since $(\nu q)\,(\nu y)\,(!y(r).\,R) \mid S'$ and S' are bisimilar (by our original assumption), we have failed to find a contradiction. Conversely, suppose S can make an output a on some channel x to become S'. Then $(\nu q)\,(\nu y)\,(!y(r).\,R) \mid S$ can make the same output and become $(\nu q)\,(\nu y)\,(!y(r).\,R) \mid S'$, where $(\nu q)\,(\nu y)\,(!y(r).\,R) \mid S'$ and S' are again bisimilar by assumption. The other two conditions in the definition of bisimilarity are checked analogously. We have therefore shown the absence of contradictions for an arbitrary R and S, and our original assumption is justified. This example allows us to give a precise meaning to the "=" at the end of the previous section: taking R to be $\bar{z}\langle r\rangle$ and S to be $\bar{z}\langle p\rangle$, we have shown that $(\nu q)\,(\nu y)\,((!y(r).\,\bar{z}\langle r\rangle) \mid \bar{z}\langle p\rangle)$ is bisimilar to $\bar{z}\langle p\rangle$.

Many variants of bisimulation have been proposed. One of the most common is so-called **weak bisimulation**, which relaxes the demand that the processes simulate each other's behavior "in lock step" and instead regards arbitrarily many steps of internal communication as equivalent to a single step. This equivalence is strictly **coarser** than the one defined above (called **strong bisimulation** when it is necessary to distinguish the two), in the sense that whenever P and Q are strongly bisimilar, they are also weakly bisimilar. In practice, weak bisimulation is often more useful, since we typically want to regard two processes as having the same observable behavior even if one consumes more processor cycles than the other.

3.4 Research Areas

The pi-calculus is just one of a large family of process calculi, differing in many details but sharing the basic orientation — focusing on interaction via communcation rather than shared variables, on describing concurrent systems using a small set of primitive operators, and on deriving useful algebraic laws for manipulating expressions written using these operators. The first historically, and among the most thoroughly studied, are Milner's *Calculus of Communicating Systems*, CCS [Milner, 1980, Milner, 1989], and Hoare's *Communicating Sequential Processes*, CSP [Hoare, 1985]. CCS is the direct predecessor of the pi-calculus: it can be described informally as the "static" fragment of the pi-calculus where the messages exchanged during communication do not contain any data (i.e. every output is of the form $\bar{x}\langle\rangle.\,P$). CSP embodies similar ideas in the form of both a theory and a programming language (Occam). Other members of the family include the variant of CCS described in Hennessy's *Algebraic Theory of Processes* [Hennessey, 1988], and Bergstra, Klop, and Baeten's systems, collectively called ACP [Baeten and Weijland, 1990]. The rapidly growing number of process calculi has led to interest in taxonomic frameworks such as Generalized Structured Operational Semantics [Groote and Vaandrager, 1992] and *action structures* [Milner, 1995], in which many different process calculi can be embedded and their properties compared.

Process calculi are widely used for specification and verification of concurrent systems, especially of communication protocols, both manually and with support from tools such as the Edinburgh Concurrency Workbench [Cleaveland *et al.*, 1993].

Numerous programming languages have combined the concurrency primitives of process calculi with more conventional features for sequential programming; well-known examples include Amber, Concurrent ML, and Facile. The "bare" pi-calculus is used the basis of the Pict language. Pi-calculus has also proved useful in the study of concurrent object-oriented languages.

4 Defining Terms

Lambda-calculus A core language of functional computation, defined in Figure 1, in which "everything is a function" and all computation proceeds by function application.

Pi-calculus A core calculus of message-based concurrency, defined in Figure 2, in which "everything is a process" and all computation proceeds by communication on channels.

Redex A subexpression in a form that is ready to be evaluated by a step of reduction.

Normal form An lambda-expression containing no redexes.

Weak head normal form A lambda-expression in which all redexes are inside the bodies of lambda-abstractions.

Confluence The property of the lambda-calculus that states that the order in which redexes are chosen for reduction does not affect the final normal form that is reached.

Combinator A lambda-expression with no free variables.

Strongly normalizing A lambda-expression for which every sequence of reductions terminates in a normal form.

Normal-order (call-by-name) reduction A reduction strategy in which the leftmost redex is always reduced first.

Applicative-order (call-by-value) reduction A reduction strategy in which the argument must be reduced to "evaluated form" before a function application can be reduced.

Contextual equivalence Formalizes the intuition that two expressions can be considered "the same" if they have the same behavior in all contexts.

Bisimulation An alternative to contextual equivalence, replacing the quantification over all contexts with the more tractable requirement that equivalent expressions should be able to match each others' behavior step-by-step.

Coinduction A proof technique associated with bisimulation. To prove that two expressions are equivalent, we assume that they are equivalent and show that no contradiction results.

References

[Baeten and Weijland, 1990] J. C. M. Baeten and W. P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[Barendregt, 1984] H. P. Barendregt. *The Lambda Calculus.* North Holland, revised edition, 1984.

[Barendregt, 1990] H. P. Barendregt. Functional programming and lambda calculus. In van Leeuwen [1990], chapter 7, pages 321–364.

[Cleaveland *et al.*, 1993] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[Davis, 1982] Martin Davis. *Computability and Unsolvability.* Dover, 1982. Previous edition 1958.

[Gordon, 1994] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, 1994.

[Groote and Vaandrager, 1992] J.F. Groote and F.W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100:202–260, 1992.

[Gunter and Scott, 1990] C. A. Gunter and D. S. Scott. Semantic domains. In van Leeuwen [1990], chapter 12, pages 633–674.

[Gunter, 1992] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* The MIT Press, Cambridge, MA, 1992.

[Hennessey, 1988] Matthew Hennessey. *Algebraic Theory of Processes.* MIT Press, 1988.

[Hindley and Seldin, 1986] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[Hoare, 1985] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Jones *et al.*, 1993] Niel D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[Milner *et al.*, 1992] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[Milner, 1980] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer Verlag, 1980.

[Milner, 1989] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Milner, 1990] R. Milner. Operational and algebraic semantics of concurrent processes. In van Leeuwen [1990], chapter 19, pages 1201–1242.

[Milner, 1991] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[Milner, 1995] Robin Milner. Calculi for interaction. *Acta Informatica*, 1995. To appear.

[Peyton Jones and Lester, 1992] Simon L. Peyton Jones and David R. Lester. *Implementing Functional Languages.* Prentice Hall, 1992.

[Plotkin, 1975] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Schmidt, 1986] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[Tennent, 1981] R. D. Tennent. *Principles of Programming Languages.* Prentice-Hall, 1981.

[van Glabbeek, 1993] R. J. van Glabbeek. The linear time – branching time spectrum II (the semantics of sequential systems with silent moves). In *Proceedings of CONCUR '93*, pages 66–81, 1993.

[van Leeuwen, 1990] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science, Volume B.* Elsevier / MIT Press, 1990.

[Winskel, 1993] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

5 Further Information

The standard text for the lambda-caclulus is Barendregt [1984]. Hindley and Seldin [1986] is less comprehensive, but somewhat more accessible. Barendregt's article in the Handbook of Theoretical Computer Science [1990] is a compact survey. Material on lambda-calculus can also be found in many textbooks on functional programming languages (e.g. [Peyton Jones and Lester, 1992]) and programming language semantics (e.g.[Schmidt, 1986, Gunter, 1992, Winskel, 1993]).

The best introduction to the pi-calculus itself is Milner's tutorial [1991]. A deeper and more accessible introduction to many of the same issues can be found in his book on CCS [Milner, 1989]. Books by Baeten and Weijland [1990], Hoare [1985], and Hennessy [1988] address other process calculi. Some semantic issues are summarized in [Milner, 1990].

New work on lambda-calculus and process calculi appears in many journals covering theoretical aspects of computer science, including *Information and Computation*, *Theoretical Computer Science*, *Mathematical Structures in Computer Science*, *Journal of Functional Programming*, and *Transactions on Programming Languages and Systems*, as well as journals in mathematics and logic. Relevant conferences include *Principles of Programming Languages (POPL)*, *Functional Programming and Computer Architecture (FPCA)*, *Lisp and Functional Programming (LFP)*, *Logic in Computer Science*, *Theoretical*

Aspects of Computer Science (TACS), the *International Conference on the Theory and Practice of Software Development (TAPSOFT)*, and the *International Conference on Concurrency Theory (CONCUR)*.