Conception, Evolution, and Application of Functional Programming Languages

PAUL HUDAK

Yale University, Department of Computer Science, New Haven, Connecticut 06520

The foundations of functional programming languages are examined from both historical and technical perspectives. Their evolution is traced through several critical periods: early work on lambda calculus and combinatory calculus, Lisp, Iswim, FP, ML, and modern functional languages such as Miranda[1] and Haskell. The fundamental premises on which the functional programming methodology stands are critically analyzed with respect to philosophical, theoretical, and pragmatic concerns. Particular attention is paid to the main features that characterize modern functional languages: higher-order functions, lazy evaluation, equations and pattern matching, strong static typing and type inference, and data abstraction. In addition, current research areas—such as parallelism, nondeterminism, input/output, and state-oriented computations—are examined with the goal of predicting the future development and application of functional languages.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classifications—*applicative languages*; *data-flow languages*; *nonprocedural languages*; *very high-level languages*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*lambda calculus and related systems*; K.2 [**History of Computing**]: Software

General Terms: Languages

Additional Key Words and Phrases: Data abstraction, higher-order functions, lazy evaluation, referential transparency, types

INTRODUCTION

The earliest programming languages were developed with one simple goal in mind: to provide a vehicle through which one could control the behavior of computers. Not surprisingly, the early languages reflected the structure of the underlying machines fairly well. Although at first blush that goal seems eminently reasonable, the viewpoint quickly changed for two very good reasons. First, it became obvious that what was easy for a machine to reason about was not necessarily easy for a human being to reason about. Second, as the number of different kinds of machines increased, the need arose for a common language with which to program all of them.

Thus from primitive assembly languages (which were at least a step up from raw machine code) there grew a plethora of high-level programming languages, beginning with FORTRAN in the 1950s. The development of these languages grew so rapidly that by the 1980s they were best characterized by grouping them into families that reflected a common computation model or programming style. Debates over which language or family of languages is best will undoubtedly persist for as long as computers need programmers.

[1] Miranda is a trademark of Research Software Ltd.

CONTENTS

The class of *functional*, or *applicative*, programming languages, in which computation is carried out entirely through the evaluation of expressions, is one such family of languages, and debates over its merits have been quite lively in recent years. Are functional languages toys? Or are they tools? Are they artifacts of theoretical fantasy or of visionary pragmatism? Will they ameliorate software woes or merely compound them? Whatever answers we might have for these questions, we cannot ignore the significant interest current researchers have in functional languages and the impact they have had on both the theory and pragmatics of programming languages in general.

Among the claims made by functional language advocates are that programs can be written quicker, are more concise, are higher level (resembling more closely traditional mathematical notation), are more amenable to formal reasoning and analysis, and can be executed more easily on parallel architectures. Of course, many of these features touch on rather subjective issues, which is one reason why the debates can be so lively.

This paper gives the reader significant insight into the very essence of functional languages and the programming methodology that they support. It starts with a discussion of the nature of functional languages, followed by an historical sketch of their development, a summary of the distinguishing characteristics of modern functional languages, and a discussion of current research areas. Through this study we will put into perspective both the power and weaknesses of the functional programming paradigm.

A Note to the Reader: This paper assumes a good understanding of the fundamental issues in programming language design and use. To learn more about modern functional programming techniques, including the important ideas behind reasoning about functional programs, refer to Bird and Wadler [1988] or Field and Harrison [1988]. To read about how to *implement* functional languages, see Peyton Jones [1987] (additional references are given in Sections 1.8 and 5).

Finally, a comment on notation: Unless otherwise stated, all examples will be written in Haskell, a recently proposed functional language standard [Hudak and Wadler 1988]. Explanations will be given in square brackets [ ] as needed.[2]

[2] Since the Haskell Report is relatively new, some minor changes to the language may occur after this paper has appeared. An up-to-date copy of the Report may be obtained from the author.

Programming Language Spectrum

Imperative languages are characterized as having an implicit *state* that is modified (i.e., side effected) by *constructs* (i.e., commands) in the source language. As a result, such languages generally have a notion of *sequencing* (of the commands) to permit precise and deterministic control over the state. Most, including the most popular, languages in existence today are imperative.

As an example, the assignment statement is a (very common) command, since its effect is to alter the underlying implicit store so as to yield a different binding for a particular variable. The begin . . . end construct is the prototypical sequencer of commands, as are the well-known goto statement (unconditional transfer of control), conditional statement (qualified sequencer), and while loop (an example of a structured command). With these simple forms, we can, for example, compute the factorial of the number x:

```
n := x;
a := 1;
while n>0 do
begin a := a*n;
      n := n-1
end;
.
.
.
```

After execution of this program, the value of a in the implicit store will contain the desired result.

In contrast, *declarative languages* are characterized as having *no* implicit state, and thus the emphasis is placed entirely on programming with *expressions* (or terms). In particular, *functional languages* are declarative languages whose underlying model of computation is the *function* (in contrast to, for example, the *relation* that forms the basis for logic programming languages).

In a declarative language state-oriented computations are accomplished by carrying the state around explicitly rather than implicitly, and looping is accomplished via recursion rather than by sequencing. For example, the factorial of x may be computed in the functional language Haskell by

```
fac x 1
where fac n a
  = if n>0 then fac (n-1) (a*n)
    else a
```

in which the formal parameters n and a are examples of carrying the state around explicitly, and the recursive structure has been arranged so as to mimic as closely as possible the looping behavior of the program given earlier. Note that the conditional in this program is an expression rather than command; that is, it denotes a value (conditional on the value of the predicate) rather than a sequencer of commands. Indeed the value of the program is the desired factorial, rather than it being found in an implicit store.

Functional (in general, declarative) programming is often described as expressing *what* is being computed rather than *how*, although this is really a matter of degree. For example, the above program may say less about how factorial is computed than the imperative program given earlier, but is perhaps not as abstract as

```
fac x
where fac n
  = if n==0 then 1
    else n*fac (n-1)
```

[== is the infix operator for equality], which appears very much like the mathematical definition of factorial and is indeed a valid functional program.

Since most languages have expressions, it is tempting to take our definitions literally and describe functional languages via derivation from conventional programming languages: Simply drop the assignment statement and any other side-effecting primitives. This approach, of course, is very misleading. The result of such a derivation is usually far less than satisfactory, since the purely functional subset of most imperative languages is hopelessly weak (although there are important exceptions, such as Scheme [Rees and Clinger 1986]).

Rather than saying what functional languages don't have, it is better to characterize them by the features they do have. For modern functional languages, those fea-

tures include higher-order functions, lazy evaluation, pattern matching, and various kinds of data abstraction—all of these features will be described in detail in this paper. Functions are treated as first-class objects, are allowed to be recursive, higher order, and polymorphic, and in general are provided with mechanisms that ease their definition and use. Syntactically, modern functional languages have an equational look in which functions are defined using mutually recursive equations and pattern matching.

This discussion suggests that what is important is the functional programming *style*, in which the above features are manifest and in which side effects are strongly discouraged but not necessarily eliminated. This is the viewpoint taken, for example, by the ML community and to some extent the Scheme community. On the other hand, there is a very large contingency of purists in the functional programming community who believe that purely functional languages are not only sufficient for general computing needs but are also better because of their "purity". At least a dozen purely functional languages exist along with their implementations.[3] The main property that is lost when side effects are introduced is *referential transparency*; this loss in turn impairs equational reasoning, as described below.

Referential Transparency and Equational Reasoning

The emphasis on a pure declarative style of programming is perhaps the hallmark of the functional programming paradigm. The term *referentially transparent* is often used to describe this style of programming, in which "equals can be replaced by equals". For example, consider the (Haskell) expression

$$\ldots x+x \ldots$$
where $x = f\ a$

The function application $(f\ a)$ may be substituted for any free occurrence of x in the scope created by the where expression, such as in the subexpression $x+x$. The same cannot generally be said of an imperative language, where we must first be sure that no assignment to x is made in any of the statements intervening between the initial definition of x and one of its subsequent uses.[4] In general this can be quite a tricky task, for example, in the case in which procedures are allowed to induce nonlocal changes to lexically scoped variables.

Although the notion of referential transparency may seem like a simple idea, the clean equational reasoning that it allows is very powerful, not only for reasoning formally about programs but also informally in writing and debugging programs. A program in which side effects are minimized but not eliminated may still benefit from equational reasoning, although naturally more care must be taken when applying such reasoning. The degree of care, however, may be much higher than we might think at first: Most languages that allow minor forms of side effects do not minimize their locality lexically—thus any call to any function in any module might conceivably introduce a side effect, in turn invalidating many applications of equational reasoning.

The perils of side effects are appreciated by the most experienced programmers in any language, although most are loathe to give them up completely. It remains the goal of the functional programming community to demonstrate that we can do completely without side effects, without sacrificing efficiency or modularity. Of course, as mentioned earlier, the lack of side effects is not all there is to the functional programming paradigm. As we shall soon see, modern functional languages rely heavily on certain other features, most notably higher-order functions, lazy evaluation, and data abstraction.

Plan of Study

Unlike many developments in computer science, functional languages have maintained the principles on which they were

[3] This situation forms an interesting contrast with the logic programming community, where Prolog is often described as declarative (whereas Lisp is usually not), and there are very few pure logic programming languages (and even fewer implementations).

[4] In all fairness, there are logics for reasoning about imperative programs, such as those espoused by Floyd, Hoare, Dijkstra, and Wirth. None of them, however, exploits any notion of referential transparency.

founded to a surprising degree. Rather than changing or compromising those ideas, modern functional languages are best classified as embellishments of a certain set of ideals. It is a distinguishing feature of modern functional languages that they have so effectively held on to pure mathematical principles in a way shared by very few other languages.

Because of this, we can learn a great deal about functional languages simply by studying their evolution. On the other hand, such a study may fail to yield a consistent treatment of any one feature that is common to most functional languages, for it will be fractured into its manifestations in each of the languages as they were historically developed. For this reason I have taken a three-fold approach to our study:

First, Section 1 provides an historical sketch of the development of functional languages. Starting with the lambda calculus as the prototypical functional language, it gradually embellishes it with ideas as they were historically developed, leading eventually to a reasonable technical characterization of modern functional languages.

Next, Section 2 presents a detailed discussion of four important concepts—higher-order functions, lazy evaluation, data abstraction mechanisms, and equations/pattern matching—which are critical components of all modern functional languages and are best discussed as independent topics.

Section 3 discusses more advanced ideas and outlines some critical research areas. Then to round out the paper, Section 4 puts some of the limitations of functional languages into perspective by examining some of the myths that have accompanied their development.

1. EVOLUTION OF FUNCTIONAL LANGUAGES

1.1 Lambda Calculus

The development of functional languages has been influenced from time to time by many sources, but none is as paramount nor as fundamental as the work of Church [1932–1933, 1941] on the *lambda calculus*. Indeed the lambda calculus is usually regarded as the first functional language, although it was certainly not thought of as programming language at the time, given that there were no computers on which to run the programs. In any case, modern functional languages can be thought of as (nontrivial) embellishments of the lambda calculus.

It is often thought that the lambda calculus also formed the foundation for Lisp, but this in fact appears not to be the case [McCarthy 1978]. The impact of the lambda calculus on early Lisp development was minimal, and it has only been very recently that Lisp has begun to evolve more toward lambda calculus ideals. On the other hand, Lisp had a significant impact on the subsequent development of functional languages, as will be discussed in Section 1.2.

Church's work was motivated by the desire to create a *calculus* (informally, a syntax for terms and set of rewrite rules for transforming terms) that captured one's intuition about the behavior of *functions*. This approach is counter to the consideration of functions as, for example, sets (more precisely, sets of argument/value pairs), since the intent was to capture the *computational* aspects of functions. A calculus is a formal way for doing just that.

Church's lambda calculus was the first suitable treatment of the computational aspects of functions. Its type-free nature yielded a particularly small and simple calculus, and it had one very interesting property, capturing functions in their fullest generality: Functions could be applied to themselves. In most reasonable theories of functions as sets, this is impossible, since it requires the notion of a set containing itself, resulting in well-known paradoxes. This ability of self-application is what gives the lambda calculus its power. It allows us to gain the effect of recursion without explicitly writing a recursive definition. Despite this powerful ability, the lambda calculus is consistent as a mathematical system—no contradictions or paradoxes arise.

Because of the relative importance of the lambda calculus to the development of functional languages, I will describe it in

detail in the remainder of this section, using modern notational conventions.

1.1.1 Pure Untyped Lambda Calculus

The abstract syntax of the *pure untyped lambda calculus* (a name chosen to distinguish it from other versions developed later) embodies what are called *lambda expressions*, defined by[5]

$x \in Id$ Identifiers
$e \in Exp$ Lambda expressions

where $e ::= x \mid e_1\ e_2 \mid \lambda x.e$

Expressions of the form $\lambda x.e$ are called *abstractions* and of the form $(e_1\ e_2)$ are called *applications*. It is the former that captures the notion of a function and the latter that captures the notion of application of a function. By convention, application is assumed to be left associative, so that $(e_1\ e_2\ e_3)$ is the same as $((e_1\ e_2)\ e_3)$.

The rewrite rules of the lambda calculus depend on the notion of substitution of an expression e_1 for all free occurrences of an identifier x in an expression e_2, which we write as $[e_1/x]e_2$.[6] Most systems, including both the lambda calculus and predicate calculus, that use substitution on identifiers must be careful to avoid name conflicts. Thus, although the intuition behind substitution is strong, its formal definition can be somewhat tedious.

To understand substitution, we must first understand the notion of the free variables of an expression e, which we write as $fv(e)$ and define by the following simple rules:

$$\begin{aligned} fv(x) &= \{x\} \\ fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) \\ fv(\lambda x.e) &= fv(e) - \{x\} \end{aligned}$$

We say that x is free in e iff $x \in fv(e)$. The substitution $[e_1/x]e_2$ is then defined inductively by

$$[e/x_i]x_j = \begin{cases} e, & \text{if } i = j \\ x_j, & \text{if } i \neq j \end{cases}$$

$$[e_1/x](e_2\ e_3) = ([e_1/x]e_2)([e_1/x]e_3)$$

$$[e_1/x_i](\lambda x_j.e_2)$$

$$= \begin{cases} \lambda x_j.e_2, & \text{if } i = j \\ \lambda x_j.[e_1/x_i]e_2, & \text{if } i \neq j \text{ and } x_j \notin fv(e_1) \\ \lambda x_k.[e_1/x_i]([x_k/x_j]e_2), & \text{otherwise,} \\ & \text{where } k \neq i,\ k \neq j, \\ & \text{and } x_k \notin fv(e_1) \cup fv(e_2) \end{cases}$$

The last rule is the subtle one, since it is where a name conflict could occur and is resolved by making a name change. The following example demonstrates application of all three rules:

$$[y/x]((\lambda y.x)(\lambda x.x)x) \equiv (\lambda z.y)(\lambda x.x)y$$

To complete the lambda calculus, we define three simple rewrite rules on lambda expressions:

(1) α-conversion (renaming): ·

$$\lambda x_i.e \Leftrightarrow \lambda x_j.[x_j/x_i]e, \quad \text{where } x_j \notin fv(e).$$

(2) β-conversion (application):

$$(\lambda x.e_1)e_2 \Leftrightarrow [e_2/x]e_1.$$

(3) η-conversion:

$$\lambda x.(e\ x) \Leftrightarrow e, \quad \text{if } x \notin fv(e).$$

These rules, together with the standard equivalence relation rules for reflexivity, symmetricity, and transitivity, induce a theory of *convertibility* on the lambda calculus, which can be shown to be consistent

[5] The notation $d \in D$ means that d is a typical element of the set D, whose elements may be distinguished by subscripting. In the case of identifiers, we assume that each x_i is unique; that is, $x_i \neq x_j$ if $i \neq j$. The notation $d ::= alt1 \mid alt2 \mid \cdots \mid altn$ is standard BNF syntax.

[6] In denotational semantics the notation $e[v/x]$ is used to denote the function e' that is just like e except that $e'\ x = v$. Our notation of placing the brackets in front of the expression is to emphasize that $[v/x]e$ is a syntactic transformation on the expression e itself.

as a mathematical system.[7] The well-known Church–Rosser theorem [Church and Rosser 1936] (actually two theorems) is what embodies the strongest form of consistency and has to do with a notion of *reduction*, which is the same as convertibility but restricted so that β-conversion and η-conversion only happen in one direction:

(1) β-reduction:

$$(\lambda x.e_1)e_2 \Rightarrow [e_2/x]e_1.$$

(2) η-reduction:

$$\lambda x.(e\ x) \Rightarrow e, \quad \text{if } x \notin fv(e).$$

We write $e_1 \overset{*}{\Rightarrow} e_2$ if e_2 can be derived from zero or more β- or η-reductions or α-conversions; in other words $\overset{*}{\Rightarrow}$ is the reflexive, transitive closure of $\Rightarrow$ including α-conversions. Similarly, $\overset{*}{\Leftrightarrow}$ is the reflexive, transitive closure of $\Leftrightarrow$. In summary, $\overset{*}{\Rightarrow}$ captures the notion of reducibility, and $\overset{*}{\Leftrightarrow}$ captures the notion of intraconvertibility.

Definition

A lambda expression is in *normal form* if it cannot be further reduced using β- or η-reduction.

Note that some lambda expressions have no normal form, such as

$$(\lambda x.\ (x\ x))\ (\lambda x.\ (x\ x)),$$

where the only possible reduction leads to an identical term, and thus the reduction process is nonterminating.

Nevertheless, the normal form appears to be an attractive canonical form for a term, has a clear sense of finality in a computational sense, and is what we intuitively think of as the value of an expression. Obviously, we would like for that value to be unique; and we would like to be able to find it whenever it exists. The Church–Rosser theorems give us positive results for both of these desires.

1.1.2 Church–Rosser Theorems

Church–Rosser Theorem I

If $e_0 \overset{*}{\Leftrightarrow} e_1$ then there exists an e_2 such that $e_0 \overset{*}{\Rightarrow} e_2$ and $e_1 \overset{*}{\Rightarrow} e_2$.[8]

In other words, if e_0 and e_1 are intraconvertible, then there exists a third term (possibly the same as e_0 or e_1) to which they can both be reduced.

Corollary

No lambda expression can be converted to two distinct normal forms (ignoring differences due to α-conversion).

One consequence of this result is that how we arrive at the normal form does not matter; that is, the order of evaluation is irrelevant (this has important consequences for parallel evaluation strategies). The question then arises as to whether or not it is always possible to find the normal form (assuming it exists). We begin with some definitions.

Definition

A *normal-order reduction* is a sequential reduction in which, whenever there is more than one reducible expression (called a *redex*), the leftmost one is chosen first. In contrast, an *applicative-order reduction* is a sequential reduction in which the leftmost innermost redex is chosen first.

Church–Rosser Theorem II

If $e_0 \overset{*}{\Rightarrow} e_1$ and e_1 is in normal form, then there exists a normal-order reduction from e_0 to e_1.

[7] The lambda calculus as we have defined it here is what Barendregt [1984] calls the $\lambda K\eta$-calculus and is slightly more general than Church's original λK-calculus (which did not include η-conversion). Furthermore, Church originally showed the consistency of the λI-calculus [Church 1941], an even smaller subset (it only allowed abstraction of x from e if x was free in e). We will ignore the subtle differences between these calculi—our version is the one most often discussed in the literature on functional languages.

[8] Church and Rosser's original proofs of their theorems are rather long, and many have tried to improve on them since. The shortest proof I am aware of for the first theorem is fairly recent and aptly due to Rosser [1982].

This is a very satisfying result; it says that if a normal form exists, we can always find it; that is, just use normal-order reduction. To see why applicative-order reduction is not always adequate, consider the following example:

Applicative-order reduction

$$\begin{aligned} & (\lambda x.\ y)((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)) \\ \Rightarrow\ & (\lambda x.\ y)((\lambda x.\ x\ x)(\lambda x.\ x\ x)) \\ \Rightarrow\ & \vdots \end{aligned}$$

Normal-order reduction

$$\begin{aligned} & (\lambda x.\, y)\ ((\lambda x.\, x\, x)\ (\lambda x.\, x\, x)) \\ \Rightarrow\ & y \end{aligned}$$

We will return to the trade-offs between normal- and applicative-order reduction in Section 2.2. For now we simply note that the strongest completeness and consistency results have been achieved with normal-order reduction.

In actuality, one of Church's (and others') motivations for developing the lambda calculus in the first place was to form a foundation for all of mathematics (in the way that, for example, set theory is claimed to provide such a foundation). Unfortunately, all attempts to extend the lambda calculus sufficiently to form such a foundation failed to yield a consistent theory. Church's original extended system was shown inconsistent by the Kleene–Rosser paradox [Kleene and Rosser 1935]; a simpler inconsistency proof is embodied in what is known as the Curry paradox [Rosser 1982]. The only consistent systems that have been derived from the lambda calculus are much too weak to claim as a foundation for mathematics, and the problem remains open today.

These inconsistencies, although disappointing in a foundational sense, did not slow down research on the lambda calculus, which turned out to be quite a nice model of functions and of computation in general. The Church–Rosser theorem was an extremely powerful consistency result for a computation model, and in fact rewrite systems completely different from the lambda calculus are often described as "possessing the Church–Rosser property" or even anthropomorphically as being Church–Rosser.

1.1.3 Recursion, λ-Definability, and Church's Thesis

Another nice property of the lambda calculus is embodied in the following theorem:

Fixpoint Theorem

Every lambda expression e has a fixpoint e' such that $(e\ e') \overset{*}{\Leftrightarrow} e'$.

Proof. Take e' to be $(Y\ e)$, where Y, known as the *Y combinator*, is defined by

$$Y \equiv \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Then we have

$$\begin{aligned} (Ye) &= (\lambda x.e(x\ x))(\lambda x.e(x\ x)) \\ &= e((\lambda x.e(x\ x))(\lambda x.e(x\ x))) \\ &= e(Y\ e) \end{aligned}$$

This surprising theorem (and equally surprising simple proof) is what has earned Y the name "paradoxical combinator". The theorem is quite significant—it means that any recursive function may be written nonrecursively (and nonimperatively). To see how, consider a recursive function f defined by

$$f \equiv \cdots f \cdots$$

This could be rewritten as

$$f \equiv (\lambda f.\ \cdots f \cdots)f$$

where the inner occurrence of f is now bound. This equation essentially says that f is a fixpoint of the lambda expression $(\lambda f.\ \cdots f \cdots)$. But that is exactly what Y computes for us, so we arrive at the following nonrecursive definition for f:

$$f \equiv Y(\lambda f.\ \cdots f \cdots)$$

As a concrete example, the factorial function

$$\begin{aligned} & fac \equiv \lambda n. \\ & \quad if\ (n = 0)\ then\ 1\ else\ (n * fac(n - 1)) \end{aligned}$$

can be written nonrecursively as

$$\begin{aligned} & fac \equiv Y(\lambda fac.\ \lambda n. \\ & \quad if\ (n = 0)\ then\ 1\ else\ (n * fac(n - 1))) \end{aligned}$$

The ability of the lambda calculus to simulate recursion in this way is the key to its power and accounts for its persistence as a useful model of computation. Church recognized this power, and is perhaps best expressed in his now famous thesis:

Church's Thesis

Effectively computable functions from positive integers to positive integers are just those definable in the lambda calculus.

This is quite a strong claim. Although the notion of functions from positive integers to positive integers can be formalized precisely, the notion of effectively computable cannot; thus no proof can be given for the thesis. It gained support, however, from Kleene [1936] who in 1936 showed that λ-definability was precisely equivalent to Gödel and Herbrand's notions of recursiveness. Meanwhile, Turing [1936] had been working on his now famous Turing machine, and in 1937 [Turing 1937] he showed that Turing computability was also precisely equivalent to λ-definability. These were quite satisfying results.[9]

The lambda calculus and the Turing machine were to have profound impacts on programming languages and computational complexity,[10] respectively, and computer science in general. This influence was probably much greater than Church or Turing could have imagined, which is perhaps not surprising given that computers did not even exist yet.

In parallel with the development of the lambda calculus, Schönfinkel and Curry were busy founding *combinatory logic*. It was Schönfinkel [1924] who discovered the surprising result that any function could be expressed as the composition of only two simple functions, K and S. Curry [1930] proved the consistency of a pure combinatory calculus, and with Feys [Curry and Feys 1958] elaborated the theory considerably. Although this work deserves as much attention from a logician's point of view as the lambda calculus, and in fact its origins predate that of the lambda calculus, we will not pursue it here since it did not contribute directly to the development of functional languages in the way that the lambda calculus did. On the other hand, the combinatory calculus was eventually to play a surprising role in the implementation of functional languages, beginning with Turner [1979] and summarized in Peyton Jones [1987, Chapter 16].

Another noteworthy attribute of the lambda calculus is its restriction to functions of one argument. That it suffices to consider only such functions was first suggested by Frege in 1893 [van Heijenoort 1967] and independently by Schönfinkel in 1924. This restriction was later exploited by Curry and Feys [1958], who used the notation $(f\ x\ y)$ to denote $((f\ x)\ y)$, which previously would have been written $f(x, y)$. This notation has become known as *currying*, and f is said to be a *curried function*. As we will see, the notion of currying has carried over today as a distinguishing syntactic characteristic of modern functional languages.

There are several variations and embellishments of the lambda calculus. They will be mentioned in the discussion of the point at which functional languages exhibited similar characteristics. In this way we can clearly see the relationship between the lambda calculus and functional languages.

1.2 Lisp

A discussion of the history of functional languages would certainly be remiss if it did not include a discussion of Lisp, beginning with McCarthy's seminal work in the late 1950s.

Although lambda calculus is often considered as the foundation of Lisp, by McCarthy's [1978] own account the lambda calculus actually played a rather small role.

---

[9] Much later Post [1943] and Markov [1951] proposed two other formal notions of effective computability; these also were shown to be equivalent to λ-definability.

[10] Although the lambda calculus and the notion of λ-definability predated the Turing machine, complexity theorists latched onto the Turing machine as their fundamental measure of decidability. This is probably because of the appeal of the Turing machine as a machine, giving it more credibility in the emerging arena of electronic digital computers. See Trakhtenbrot [1988] for an interesting discussion of this issue.

Its main impact came through McCarthy's desire to represent functions anonymously, and Church's λ-notation was what he chose: A lambda abstraction written $\lambda x.e$ in lambda calculus would be written (lambda (x) e) in Lisp.

Beyond that, the similarity wanes. For example, rather than use the Y combinator to express recursion, McCarthy invented the *conditional expression*[11] with which recursive functions could be defined explicitly (and, arguably, more intuitively). As an example, the nonrecursive factorial function given in the lambda calculus in Section 1.1.3 would be written recursively in Lisp in the following way:

```
(define fac (n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))
```

This and other ideas were described in two landmark papers in the early 1960s [McCarthy 1960; 1963] that inspired work on Lisp for many years to come.

McCarthy's original motivation for developing Lisp was the desire for an algebraic list-processing language for use in artificial intelligence research. Although symbolic processing was a fairly radical idea at the time, his aims were quite pragmatic. One of the earliest attempts at designing such a language was suggested by McCarthy and resulted in FLPL (FORTRAN-compiled list processing language), implemented in 1958 on top of the FORTRAN system on the IBM 704 [Gelernter et al. 1960]. During the next few years McCarthy designed, refined, and implemented Lisp. His chief contributions during this period were the following:

(1) The conditional expression and its use in writing recursive functions.
(2) The use of lists and higher-order operations over lists such as mapcar.
(3) The central idea of a cons cell and the use of garbage collection as a method of reclaiming unused cells.
(4) The use of S-expressions (and abstract syntax in general) to represent both program and data.[12]

All four of these features are essential ingredients of any Lisp implementation today; the first three are essential to functional language implementations as well.

A simple example of a typical Lisp definition is the following one for mapcar:

```
(define mapcar (fun lst)
  (if (null lst)
      nil
      (cons (fun (car lst))
            (mapcar fun (cdr
            lst)))))
```

This example demonstrates all of the points mentioned above. Note that the function fun is passed as an argument to mapcar. Although such higher-order programming was very well known in lambda calculus circles, it was certainly a radical departure from FORTRAN and has become one of the most important programming techniques in Lisp and functional programming (higher order functions are discussed more in Section 2.1). The primitive functions cons, car, cdr, and null are the well-known operations on lists whose names are still used today. cons creates a new list cell without burdening the user with explicit storage management; similarly, once that cell is no longer needed a "garbage collector" will come along and reclaim it, again without user involvement. For example, since mapcar constructs a new list from an old one, in the call

```
(mapcar fun (cons a (cons b nil)))
```

the list (cons a (cons b nil)) will become garbage after the call and will automatically be reclaimed. Lists were to become the paradigmatic data structure in Lisp and early functional languages.

The definition of mapcar in a modern functional language such as Haskell would appear similarly, except that pattern matching would be used to destructure the

[11] The conditional in FORTRAN (essentially the only other programming language in existence at the time) was a statement, not an expression, and was for control, not value-defining, purposes.

[12] Interestingly, McCarthy [1978] claims that it was the *read* and *print* routines that influenced this notation most.

list:

```
mapcar fun [ ]     = [ ]
mapcar fun (x:xs) = fun x : mapcar fun xs
```

[[ ] is the null list and : is the infix operator for cons; also note that function application has higher precedence than any infix operator.]

McCarthy was also interested in designing a practical language, and thus Lisp had many pragmatic features—in particular, sequencing, the assignment statement, and other primitives that induced side effects on the store. Their presence undoubtedly had much to do with early experience with FORTRAN. Nevertheless, in his early papers, McCarthy emphasized the mathematical elegance of Lisp, and in a much later paper his student Cartwright demonstrated the ease with which one could prove properties about pure Lisp programs [Cartwright 1976].

Despite its impurities, Lisp had a great influence on functional language development, and it is encouraging to note that modern Lisps (especially Scheme) have returned more to the purity of the lambda calculus rather than the ad hocery that plagued the Maclisp era. This return to purity includes the first-class treatment of functions and the lexical scoping of identifiers. Furthermore, the preferred modern style of Lisp programming, such as espoused by Abelson et al. [1985], can be characterized as being predominantly side-effect free. And, finally, note that Henderson's [1980] Lispkit Lisp is a purely functional version of Lisp that uses an infix, algebraic syntax.

1.2.1 Lisp in Retrospect

Before continuing the historical development it is helpful to consider some of the design decisions McCarthy made and how they would be formalized in terms of the lambda calculus. It may seem that conditional expressions, for example, are an obvious feature to have in a language, but that only reflects our familiarity with modern high-level programming languages, most of which have them. In fact the lambda calculus version of the factorial example given in the previous section used a conditional (not to mention arithmetic operators), yet most readers probably understood it perfectly and did not object to the departure from precise lambda calculus syntax.

The effect of conditional expressions can in fact be achieved in the lambda calculus by encoding the true and false values as functions, as well as by defining a function to emulate the conditional:

$$\begin{aligned} \mathit{true} &\equiv \lambda x.\lambda y.x \\ \mathit{false} &\equiv \lambda x.\lambda y.y \\ \mathit{cond} &\equiv \lambda p.\lambda c.\lambda a.(p\ c\ a) \end{aligned}$$

In other words, $(\mathit{cond}\ p\ c\ a) \equiv (\mathit{if}\ p\ \mathit{then}\ c\ \mathit{else}\ a)$. One can then define, for example, the factorial function by

$$\mathit{fac} \equiv \lambda n.\ \mathit{cond}\ (=\ n\ 0)\ 1\ (*\ n(\mathit{fac}\ (-\ n\ 1)))$$

where = is defined by

$$\begin{aligned} (=\ n\ n) &\Rightarrow \mathit{true} \\ (=\ n\ m) &\Rightarrow \mathit{false}, \quad \mathit{if}\ m \neq n \end{aligned}$$

where m and n range over the set of integer constants. However, I am still cheating a bit by not explaining the nature of the objects −, *, 0, 1, and so on, in pure lambda calculus terms. It turns out that they can be represented in a variety of ways, essentially using functions to simulate the proper behavior, just as for true, false, and the conditional (for the details, see Church [1941]). In fact *any* conventional data or control structure can be simulated in the lambda calculus; if this were not the case, it would be difficult to believe Church's thesis.

Even if McCarthy knew of these ways to express things in the lambda calculus (there is reason to believe that he did not), efficiency concerns might have rapidly led him to consider other alternatives, especially since FORTRAN was the only high-level programming language with which anyone had any experience. In particular, FORTRAN functions evaluated their arguments *before* entering the body of the function, resulting in what is often called a *strict*, or *call-by-value*, evaluation policy, corresponding roughly to applicative-order reduction in the lambda calculus. With this strategy extended to the primitives, including the conditional, we cannot easily define

recursive functions. For example, in the above definition of factorial all three arguments to *cond* would be evaluated, including *fac* $(-\ n\ 1)$, resulting in nontermination.

Nonstrict evaluation, corresponding to the normal-order reduction that is essential to the lambda calculus in realizing recursion, was not very well understood at the time—it was not at all clear how to implement it efficiently on a conventional von Neumann computer—and we would have to wait another 20 years or so before such an implementation was even attempted.[13] The conditional expression essentially allowed one to invoke normal-order, or nonstrict, evaluation selectively. Stated another way, McCarthy's conditional, although an expression, was compiled into code that essentially controlled the reduction process. Most imperative programming languages today that allow recursion do just that, and thus even though such languages are often referred to as strict, they all rely critically on at least one nonstrict construct: the conditional.

1.2.2 Lambda Calculus with Constants

The conditional expression is actually only one example of very many primitive functions that were included in Lisp. Rather than explain them in terms of the lambda calculus by a suitable encoding (i.e., compilation), it is perhaps better to extend the lambda calculus formally by adding a set of constants along with a set of what are usually called *δ-rules*, which state relationships between constants and effectively extend the basis set of α-, β-, and η-reduction rules. For example, the reduction rules for = given earlier (and repeated below) are δ-rules. This new calculus, often called the *lambda calculus with constants*, can be given a precise abstract syntax:

$x \in Id$ Identifiers
$c \in Con$ Constants
$e \in Exp$ Lambda expressions

where $e ::= x \mid c \mid e_1\ e_2 \mid \lambda x.e$

for which various δ-rules apply, such as the following:

$$(=\ 0\ 0) \Rightarrow True$$
$$(=\ 0\ 1) \Rightarrow False$$
$$\vdots$$
$$(+\ 0\ 0) \Rightarrow 0$$
$$(+\ 0\ 1) \Rightarrow 1$$
$$\vdots$$
$$(+\ 27\ 32) \Rightarrow 59$$
$$\vdots$$
$$(If\ True\ e_1\ e_2) \Rightarrow e_1$$
$$(If\ False\ e_1\ e_2) \Rightarrow e_2$$
$$\vdots$$
$$(Car\ (Cons\ e_1\ e_2)) \Rightarrow e_1$$
$$(Cdr\ (Cons\ e_1\ e_2)) \Rightarrow e_2$$
$$\vdots$$

where =, +, 0, 1, *If*, *True*, *False*, *Cons*, and so on, are elements of *Con*.

The above rules can be shown to be a *conservative extension* of the lambda calculus, a technical term that in our context essentially means that convertible terms in the original system are still convertible in the new, and (perhaps more importantly) inconvertible terms in the original system are still inconvertible in the new. In general, care must be taken when introducing δ-rules, since all kinds of inconsistencies could arise. For a quick and dirty example of inconsistency, we might define a primitive function over integers called *broken* with the following δ-rules:

$$(broken\ 0) \Rightarrow 0$$
$$(broken\ 0) \Rightarrow 1$$

which immediately implies that more than one normal form exists for some terms, violating the first Church–Rosser theorem (see Section 1.1).

[13] On the other hand, the call-by-name evaluation strategy invented in ALGOL had very much of a normal-order reduction flavor. See Wadsworth [1971] and Wegner [1968] for early discussions of these issues.

As a more subtle example, suppose we define the logical relation *Or* by

$$(Or\ True\ e) \Rightarrow True$$
$$(Or\ e\ True) \Rightarrow True$$
$$(Or\ False\ False) \Rightarrow False$$

Although these rules form a conservative extension of the lambda calculus, a commitment to evaluate either of the arguments may lead to nontermination (even if the other argument may reduce to *True*). In fact, it can be shown that with the above rules there does not exist a deterministic sequential reduction strategy that will guarantee that the normal form *True* will be found for all terms having such a normal form, and thus the second Church–Rosser property is violated. This version of *Or* is often called the *parallel or*, since a parallel reduction strategy is needed to implement it properly (and with which the first Church–Rosser theorem will at least hold). Alternatively, we could define a sequential or by

$$(Or\ True\ e) \Rightarrow True$$
$$(Or\ False\ e) \Rightarrow e$$

which can be shown to satisfy both Church–Rosser theorems.

1.3 Iswim

Historically speaking, Peter Landin's work in the mid 1960s was the next significant impetus to the functional programming paradigm. Landin's work was deeply influenced by that of Curry and Church. His early papers discussed the relationship between lambda calculus and both machines and high-level languages (specifically ALGOL 60). Landin [1964] discussed how one could mechanize the evaluation of expressions through an abstract machine called the SECD machine; in Landin [1965] he formally defined a nontrivial subset of ALGOL 60 in terms of the lambda calculus.

It is apparent from his work that Landin regarded highly the expressiveness and purity of the lambda calculus and at the same time recognized its austerity. Undoubtedly as a result of this work, in 1966 Landin introduced a language (actually a family of languages) called Iswim (for If You See What I Mean), which included a number of significant syntactic and semantics ideas [Landin 1966]. Iswim, according to Landin, "can be looked on as an attempt to deliver Lisp from its eponymous commitment to lists, its reputation for hand-to-mouth storage allocation, the hardware dependent flavor of its pedagogy, its heavy bracketing, and its compromises with tradition." When all is said and done, the primary contributins of Iswim, with respect to the development of functional languages, are the following:

(1) Syntactic innovations
   (a) The abandonment of prefix syntax in favor of infix.
   (b) The introduction of *let* and *where* clauses, including a notion of simultaneous and mutually recursive definitions.
   (c) The use of an off-side rule based on indentation rather than separators (such as commas or semicolons) to scope declarations and expressions. For example (using Haskell syntax), the program fragment

```
e where f x = x
        a b = 1
```

   cannot be confused with

```
e where f x = x a
        b = 1
```

   and is equivalent to what might otherwise be written as

```
e where {f x = x; a b = 1}
```

(2) Semantic innovations
   (a) An emphasis on generality. Landin was half serious in hoping that the Iswim family could serve as the "next 700 programming languages." Central to his strategy was the idea of defining a syntactically rich language in terms of a very small but expressive core language.
   (b) An emphasis on equational reasoning (i.e., the ability to replace equals with equals). This elusive idea was backed up with four sets of rules for reasoning about expressions, declarations, primitives, and problem-oriented extensions.

(c) The SECD machine as a simple abstract machine for executing functional programs.

We can think of Landin's work as extending the lambda calculus with constants defined in the last section so as to include more primitives, each with its own set of δ-rules, but more importantly *let* and *where* clauses, for which it is convenient to introduce the syntactic category of declarations:

$$e \in Exp \qquad \text{Expressions}$$

$$\text{where } e ::= \cdots \mid e \text{ where } d_1 \cdots d_n \mid \textit{let } d_1 \cdots d_n \textit{ in } e$$

$$d \in Decl \qquad \text{Declarations}$$

$$\text{where } d ::= x = e \mid x\ x_1 \cdots x_n = e$$

and for which we then need to add some axioms (i.e., reduction rules) to capture the desired semantics. Landin proposed special constructs for defining simultaneous and mutually recursive definitions, but we will take a simpler and more general approach here: We assume that a block of declarations $d_1 \cdots d_n$ always is potentially mutually recursive—if it isn't, our rules still work:

$$(\textit{let } d_1 \cdots d_n \textit{ in } e) \Rightarrow (e \text{ where } d_1 \cdots d_n)$$

$$(x\ x_1 \cdots x_n = e) \Rightarrow (x = \lambda x_1.\lambda x_2. \cdots \lambda x_n.e)$$

$$(e \text{ where } x_1 = e_1) \Rightarrow (\lambda x_1.e)(Y\lambda x_1.e_1)$$

$$(e \text{ where } (x_1 = e_1) \cdots (x_n = e_n)) \Rightarrow (\lambda x_1.e)(Y\lambda x_1.e_1) \text{ where } x_2 = (\lambda x_1.e_2)(Y\lambda x_1.e_1)$$

$$\vdots$$

$$x_n = (\lambda x_1.\ e_n)(Y\lambda x_1.e_1)$$

These rules are semantically equivalent to Landin's, but they avoid the need for a tupling operator to handle mutual recursion and they use the Y combinator (defined in Section 1.1) instead of an iterative unfolding step.

We will call this resulting system the *recursive lambda calculus with constants*, or just *recursive lambda calculus*.

Landin's emphasis on expressing *what* the desired result is, as opposed to saying *how* to get it, and his claim that Iswim's declarative[14] style of programming was better than the incremental and sequential imperative style were ideas to be echoed by functional programming advocates to this day. On the other hand, it took another 10 years before interest in functional languages was to be substantially renewed. One of the reasons is that there were no decent implementations of Iswim-like languages around; this reason, in fact, was to persist into the 1980s.

1.4 APL

Iverson's [1962] APL, although not a purely functional programming language, is worth mentioning because its functional subset is an example of how we could achieve functional programming without relying on lambda expressions. In fact, Iverson's design of APL was motivated out of his desire to develop an algebraic programming language for arrays, and his original work used an essentially functional notation. Subsequent development of APL resulted in several imperative features, but the underlying principles should not be overlooked.

APL was also unique in its goal of succinctness, in that it used a specially designed alphabet to represent programs—each letter corresponding to one operator. That APL became popular is apparent in the fact that many keyboards, both for typewriters and computer terminals, carried the APL alphabet. Backus' FP, which came after APL, was certainly influenced by the APL philosophy, and its abbreviated publication form also used a specialized alphabet (see the example in Section 1.5). In fact FP has much in common with APL, the primary difference being that FP's fundamental data structure is the *sequence*, whereas APL's is the *array*.

It is worth noting that recent work on APL has revived some of APL's purely functional foundations. The most notable

[14] Landin actually disliked the term "declarative," preferring instead "denotative."

work is that of Tu [Tu 1986; Tu and Perlis 1986], who designed a language called FAC, for Functional Array Calculator (presumably a take off on Turner's KRC, Kent Recursive Calculator). FAC is a purely functional language that adopts most of the APL syntax and programming principles but also has special features to allow programming with infinite arrays; naturally, lazy evaluation plays a major role in the semantics of the language. Another interesting approach is that of Mullin [1988].

1.5 FP

Backus' FP was one of the earliest functional languages to receive widespread attention. Although most of the features in FP are not found in today's modern functional languages, Backus' [1978] Turing Award lecture was one of the most influential and now most-often cited papers extolling the functional programming paradigm. It not only stated quite eloquently why functional programming was "good" but also quite vehemently why traditional imperative programming was "bad".[15] Backus' coined the term "word-at-a-time programming" to capture the essence of imperative languages, showed how such languages were inextricably tied to the von Neumann machine, and gave a convincing argument why such languages were not going to meet the demands of modern software development. That this argument was being made by the person who is given the most credit for designing FORTRAN and who also had significant influence on the development of ALGOL led substantial weight to the functional thesis. The exposure given to Backus' paper was one of the best things that could have happened to the field of functional programming, which at the time was certainly not considered mainstream.

Despite the great impetus Backus' paper gave to functional programming, it is interesting to note that in the same paper Backus also said that languages based on lambda calculus would have problems, both in implementation and expressiveness, because the model was not suitably history sensitive (more specifically, it did not handle large data structures such as databases very easily). With regard to implementation, this argument is certainly understandable because it was not clear how to implement the notion of substitution in an efficient manner nor was it clear how to structure data in such ways that large data structures could be implemented efficiently (both of these issues are much better understood today). With regard to expressiveness, that argument is still a matter of debate today. In any case, these problems were the motivation for Backus' Applicative State Transition (AST) Systems, in which state is introduced as something on which purely functional programs interact with in a more traditional (i.e., imperative) way.

Perhaps more surprising, and an aspect of the paper that is usually overlooked, Backus had this to say about lambda-calculus based systems:

> An FP system is founded on the use of a fixed set of combining forms called functional forms. . . . In contrast, a lambda-calculus based system is founded on the use of the lambda expression, with an associated set of substitution rules for variables, for building new functions. The lambda expression (with its substitution rules) is capable of defining all possible computable functions of all possible types and of any number of arguments. This freedom and power has its disadvantages as well as its obvious advantages. It is analogous to the power of unrestricted control statements in conventional languages: with unrestricted freedom comes chaos. If one constantly invents new combining forms to suit the occasion, as one can in the lambda calculus, one will not become familiar with the style or useful properties of the few combining forms that are adequate for all purposes.

Backus' argument, of course, was in the context of garnering support for FP, which had a small set of combining forms that were claimed to be sufficient for most programming applications. One of the advantages of this approach is that each of these combining forms could be named with particular brevity, and thus programs become quite compact—this was exactly the approach taken by Iverson in designing APL

[15] Ironically, the Turing Award was given to Backus in a large part because of his work on FORTRAN.

(see Section 1.4). For example, an FP program for inner product looks like

$$\text{Def } IP \equiv (/+) \circ (\alpha\times) \circ Trans$$

where /, $\circ$, and α are combining forms called insert, compose, and apply-to-all, respectively. In a modern functional language such as Haskell this would be written with slightly more verbosity as

```
ip l1 l2 = foldl (+) 0 (map2 (*) l1 l2)
```

[In Haskell an infix operator such as + may be passed as an argument by surrounding it in parentheses.] Here foldl is the equivalent of insert (/), and map2 is a two-list equivalent of apply-to-all (α), thus eliminating the need for *Trans*. These functions are predefined in Haskell, as they are in most modern functional languages, for the same reason that Backus argues—they are commonly used. If they were not, they could easily be defined by the user. For example, we may wish to define an infix composition operator for functions, the first of which is binary, as follows:

```
(f .°. g) x y = f (g x y)
```

[Note how infix operators may be defined in Haskell; operators are distinguished lexically by being nonalphabetic.] With this we can reclaim much of FP's succinctness in defining *ip*:

```
ip = foldl (+) 0 .°. map2 (*)
```

[Recall that in Haskell, function application has higher precedence than infix operator application.] It is for this reason, together with the fact that FP's specialization precluded the generality afforded by user-defined higher order functions (which is all that combining forms are), that modern functional languages did not follow the FP style. As we shall soon see, certain other kinds of syntactic sugar became more popular instead (such as pattern matching, list comprehensions, and sections).

Many extensions to FP have been proposed over the years, including the inclusion of strong typing and abstract datatypes [Guttag et al. 1981]. In much more recent work, Backus et al. [1986] have designed the language FL, which is strongly (although dynamically) typed and in which higher order functions and user-defined datatypes are allowed. Its authors still emphasize the algebraic style of reasoning that is paramount in FP, although it is also given a denotational semantics that is probably consistent with respect to the algebraic semantics.

1.6 ML

In the mid 1970s, at the same time Backus was working on FP at IBM, several research projects were underway in the United Kingdom that related to functional programming, most notably work at Edinburgh. There Gordon et al. [1979] had been working on a proof-generating system called LCF for reasoning about recursive functions, in particular in the context of programming languages. The system consisted of a deductive calculus called PPλ (polymorphic predicate calculus) together with an interactive programming language called ML, for metalanguage (since it served as the command language for LCF).

LCF is quite interesting as a proof system, but its authors soon found that ML was also interesting in its own right, and they proceeded to develop it as a stand-alone functional programming language [Gordon et al. 1978]. That it was, and still is, called a functional language is actually somewhat misleading, since it has a notion of *references* that are locations that can be stored into and read from, much as variables are assigned and read. Its I/O system also induces side effects and is not referentially transparent. Nevertheless, the style of programming that it encourages is still functional, and that is the way it was promoted (the same is somewhat true for Scheme, although to a lesser extent).

More recently a standardization effort for ML has taken place, in fact taking some of the good ideas of Hope [Burstall et al. 1980] (such as pattern matching) along with it, yielding a language now being called Standard ML, or SML [Milner 1984; Wikstrom 1988].

ML is a fairly complete language—certainly the most practical functional language at the time it appeared—and SML is even more so. It has higher order functions,

a simple I/O facility, a sophisticated module system, and even exceptions. But by far the most significant aspect of ML is its type system, which is manifested in several ways:

(1) It is strongly and statically typed.
(2) It uses type inference to determine the type of every expression, instead of relying on excplicit type declarations.
(3) It allows polymorphic functions and data structures; that is, functions may take arguments of arbitrary type if in fact the function does not depend on that type (similarly for data structures).
(4) It has user-defined concrete and abstract datatypes (an idea actually borrowed from Hope and not present in the initial design of ML).

ML was the first language to use type inference as a semantically integrated component of the language, and at the same time its type system was richer than any previous statically typed language in that it permitted true polymorphism. It seemed that the best of two worlds had been achieved—not only is making explicit type declarations (a sometimes burdensome task) not required, but in addition a program that successfully passes the type inferencer is guaranteed not to have any type errors. Unfortunately, this idyllic picture is not completely accurate (although it is close), as we shall soon see.

We shall next discuss the issue of types in a foundational sense, thus continuing our plan of elaborating the lambda calculus to match the language features being discussed. This will lead us to a reasonable picture of ML's Hindley–Milner type system, the rich polymorphic type system that was mentioned above and that was later adopted by every other statically typed functional language, including Miranda and Haskell. Aside from the type system, the two most novel features in ML are its references and modules, which are also covered in this section. Discussion of ML's data abstraction facilities will be postponed until Section 2.3.

1.6.1 Hindley–Milner Type System

We can introduce types into the pure lambda calculus by first introducing a domain of basic types, say *BasTyp*, as well as a domain of derived types, say *Typ*, and then requiring that every expression be tagged with a member of *Typ*, which we do by superscripting, as in e^{τ}. The derived type $\tau_2 \rightarrow \tau_1$ denotes the type of all functions from values of type τ_2 to values of type τ_1, and thus a proper application will have the form $e_1^{\tau_2 \rightarrow \tau_1} e_2^{\tau_2})\tau_1$. Modifying the pure lambda calculus in this way, we arrive at the *pure typed lambda calculus*:

$b \in$	Basic types
$\tau \in$	Derived types
	where $\tau ::= b \mid \tau_1 \rightarrow \tau_2$
$x^{\tau} \in Id$	Typed identifiers
$e \in Exp$	Typed lambda expressions
	where $e ::= x^{\tau}$
	$\mid (e_1^{\tau_2 \rightarrow \tau_1} e_2^{\tau_2})^{\tau_1}$
	$\mid (\lambda x^{\tau_2}.e^{\tau_1})^{\tau_2 \rightarrow \tau_1}$

for which we then provide the following reduction rules:

(1) Typed-α-conversion:

$$(\lambda x_1^{\tau_1} \,.\, e^{\tau}) \Leftrightarrow (\lambda x_2^{\tau_1} \,.\, [x_2^{\tau_1}/x_1^{\tau_1}]e^{\tau}),$$

where $x_2^{\tau_1} \notin fv(e^{\tau})$.

(2) Typed-β-conversion:

$$((\lambda x^{\tau_2} \,.\, e_1^{\tau_1})e_2^{\tau_2}) \Leftrightarrow [e_2^{\tau_2}/x^{\tau_2}]e_1^{\tau_1}.$$

(3) Typed-η-converson:

$$\lambda x^{\tau_1} \,.\, (e^{\tau_2} x^{\tau_1}) \Leftrightarrow e^{\tau_2}, \quad \text{if } x^{\tau_1} \notin fv(e^{\tau_2}).$$

To preserve type correctness, we assume that the typed identifiers x^{τ_1} and x^{τ_2}, where $\tau_1 \neq \tau_2$, are distinct identifiers. Note then how every expression in our new calculus carries with it its proper type, and thus type checking is built in to the syntax.

Unfortunately, there is a serious problem with this result: Our new calculus has lost the power of λ-definability. In fact, every term in the pure typed lambda calculus can be shown to be *strongly normalizable*, meaning each has a normal form, and there is an effective procedure for reducing each of them to its normal form [Fortune et al.

1985]. Consequently, we can only compute a rather small subset of functions from integers to integers—namely, the extended polynomials [Barendregt 1984].[16]

The reader can gain some insight into the problem by trying to write the definition of the Y combinator–that paradoxical entity that allowed us to express recursion—as a typed term. The difficulty lies in properly typing self-application (recall the discussion in Section 1.1), since typing (ee) requires that e have both the type $\tau_2 \rightarrow \tau_1$ and τ_2, which cannot be done within the structure we have given. It can, in fact, be shown that the pure typed lambda calculus has *no* fixpoint operator.

Fortunately, there is a clever way to solve this dilemma. Instead of relying on self-application to implement recursion, simply add to the calculus a family of constant fixpoint operators similar to Y, only typed. To do this, we first move into the *typed lambda calculus with constants*, in which a domain of constants *Con* is added as in the (untyped) lambda calculus with constants. We then include in *Con* a *typed fixpoint operator* of the form Y_τ for every type τ, where

$$Y_\tau\colon (\tau \rightarrow \tau) \rightarrow \tau$$

Then for each fixpoint operator Y_τ we add the δ-rule:

Typed-Y-conversion:

$$(Y_\tau e^{\tau\rightarrow\tau})^\tau \Leftrightarrow (e^{\tau\rightarrow\tau}(Y_\tau e^{\tau\rightarrow\tau})^\tau)^\tau$$

The reader may easily verify that type consistency is maintained by this rule.

By ignoring the type information, we can see that the above δ-rule corresponds to the conversion $(Yf) \Leftrightarrow (f(Yf))$ in the untyped case, and the same trick for implementing recursion with Y as discussed in Section 1.1 can be used here, thus regaining λ-definability. For example, a nonrecursive definition of the typed factorial function would be the same as the untyped version given earlier, except that Y_τ would be used where $\tau = Int \rightarrow Int$.

In addition to this calculus, we can derive a *typed recursive lambda calculus with constants* in the same way that we did in the untyped case, except that instead of the unrestricted Y combinator we use the typed versions defined above.

At this point our type system is about on par with that of a strongly and explicitly typed language such as Pascal or Ada. We would, however, like to do better. As mentioned in Section 1.6, the designers of ML extended the state of the art of type systems in two significant ways:

- They permitted *polymorphic* functions.
- They used *type inference* to infer types rather than requiring that they be declared explicitly.

As an example of a polymorphic function, consider

```
map :: (a → b) → [a] → [b]
map f [ ] = [ ]
map f (x:xs) = f x : map f xs
```

The first line is a *type signature* that declares the type of map; it can be read as "for all types a and b, map is a function that takes two arguments, a function from a into b and a list of elements of type a, and returns a list of elements of type b."

[Type signatures are optional in Haskell; the type system is able to infer them automatically. Also note that the type constructor $\rightarrow$ is right associative, which is consistent with the left associativity of function application.]

Therefore, map can be used to map square down a list of integers, or head down a list of lists, and so on. In a monomorphic language such as Pascal, one would have to define a separate function for each of these. The advantage of polymorphism should be clear.

One way to achieve polymorphism in our calculus is to add a domain of *type variables* and extend the domain of derived types accordingly:

$b \in BasTyp$	Basic types
$v \in TypId$	Type variables
$\tau \in Typ$	Derived types

where $\tau ::= b \mid v \mid \tau_1 \rightarrow \tau_2$

[16] On the bright side, some researchers view the strong normalization property as a *feature*, since it means that all programs are guaranteed to terminate. Indeed this property forms the basis of much of the recent work on using constructive type theory as a foundation for programming languages.

Thus, for example, $\upsilon \rightarrow \tau$ is a proper type and can be read as if the type variable was universally quantified: "for all types υ, the type $\upsilon \rightarrow \tau$." To accommodate this we must change the rule for β-conversion to read

(2) Typed-β-conversion with type variables:

(a) $((\lambda x^{\tau_2}.e_2^{\tau_2}) \Leftrightarrow ([e_2^{\tau_2}/x^{\tau_2}]e_1^{\tau_1})^{\tau_1}$

(b) $((\lambda x^{\upsilon}.e_1^{\tau_1})e_2^{\tau_2})$
$\Leftrightarrow ([\tau_2/\upsilon]([e_2^{\tau_2}/x^{\upsilon}]e_1^{\tau_1}))^{\tau_1}$

where substitution on type variables is defined in the obvious way. Note that this rule implies the validity of expressions of the form $(e_1^{\upsilon \rightarrow \tau_1} e_2^{\tau_2})^{\tau_1}$. Similar changes are required to accommodate expressions such as $(e_1^{\tau \rightarrow \upsilon} e_2^{\tau})^{\upsilon}$.

But, alas, now that type variables have been introduced, it is no longer clear whether a program is properly typed—it is not built in to the static syntactic structure of the calculus. In fact, the type-checking problem for this calculus is undecideable, being a variant of a problem known as *partial* polymorphic type inference [Boehm 1985; Pfenning 1988].

Rather than trying to deal with the type-checking problem directly, we might go one step further with our calculus and try to attain ML's lack of a requirement for explicit typing. We can achieve this by simply erasing all type annotations and then trying to solve the seemingly harder problem of inferring types of completely naked terms. Surprisingly, it is not known whether an effective type inference algorithm exists for this calculus, even though the problem of partial polymorphic type inference, known to be undecidable, seems as if it should be easier.

Fortunately, Hindley [1969] and Milner [1978] independently discovered a restricted polymorphic type system that is almost as rich as that provided by our calculus and for which type inference is decidable. In other words, there exist certain terms in the calculus presented above that one could argue are properly typed but would not be allowed in the Hindley–Milner system. The system still allows polymorphism, such as exhibited by map defined earlier, and is able to infer the type of functions such as map without any type signatures present. The use of such polymorphism however, is limited to the scope in which map was defined. For example, the program

```
silly map f g
where f     :: Int → Int
      g     :: Char → Char
      map   :: (a → b) → [a] → [b]
      silly m f g = (m f num_list,
                     m g char_list)
```

[(e1, e2) is a tuple] results in a type error, since map is passed as an argument and then instantiated in two different ways; that is, once as type (Int → Int) → [Int] → [Int] and once as type (Char → Char) → [Char] → [Char]. If map were instantiated in several ways within the scope in which it was defined or if *m* were only instantiated in one way within the function silly, there would have been no problem.

This example demonstrates a fundamental limitation to the Hindley–Milner type system, but in practice the class of programs that the system rejects is not large and is certainly smaller than that rejected by any existing type-checking algorithm for conventional languages in that, if nothing else, it allows polymorphism. Many other functional languages have since then incorporated what amounts to a Hindley–Milner type system, including Miranda and Haskell. It is beyond the scope of this article to discuss the details of type inference, but the reader may find good pragmatic discussions in Hancock [1987] and Damas and Milner [1982] and a good theoretical discussion in Milner [1978].

As a final comment we point out that an alternative way to gain polymorphism is to introduce types as values and give them at least some degree of first-class status (as we did earlier for functions); for example, allowing them to be passed as arguments and returned as results. Allowing them to be passed as arguments only (and then used in type annotations in the standard way) results in what is known as the *polymorphic* or *second-order typed lambda calculus.* Girard [1972] and Reynolds [1974] discovered and studied this type system independently, and it appears to have great expressive power. It turns out, however, to be essentially equivalent to the system we

developed earlier and has the same difficulties with respect to type inference. It is, nevertheless, an active area of current research (see Cardelli and Wegner [1985] and Reynolds [1985] for good summaries of this work).

1.6.2 ML's References

A *reference* is essentially a pointer to a cell containing values of a particular type; references are created by the (pseudo)function ref. For example, ref 5 evaluates to an integer reference—a pointer to a cell that is allowed to contain only integers and in this case having initial contents 5. The contents of a cell can be read using the prefix operator !. Thus if x is bound to ref 5 then $!x$ returns 5.

The cell pointed to by a reference may be updated via *assignment* using the infix operator :=. Continuing with the above example, $x := 10$, although an expression, has the side effect of updating the cell pointed to by x with the value 10. Subsequent evaluations of $!x$ will then return 10. Of course, to make the notion of subsequent well-defined, it is necessary to introduce sequencing constructs; indeed ML even has an iterative while construct.

References in ML amount to assignable variables in a conventional programming language and are only notable in that they can be included within a type structure such as Hindley–Milner's and can be relegated to a minor role in a language that is generally proclaimed as being functional. A proper treatment of references within a Hindley–Milner type system can be found in Tofte [1988].

1.6.3 Modules

Modules in ML are called *structures* and are essentially reified environments. The type of a structure is captured in its *signature* and contains all of the static properties of a module that are needed by some other module that might use it. The use of one module by another is captured by special functions called *functors* that map structures to new structures. This capability is sometimes called a *parameterized module* or *generic package*.

For example, a new signature called SIG (think of it as a new type) may be declared by

```
signature SIG =
  sig
    val    x : int
    val succ : int → int
  end
```

in which the types of two identifiers have been declared, x of type int and succ of type int → int. The following structure S (think of it as an environment) has the implied signature SIG defined above:

```
structure S =
  struct
    val x = 5
    val succ x = x+1
  end
```

If we then define the following functor F (think of it as a function from structures to structures):

```
functor F(T:SIG) =
  struct
    val y = T.x + 1
    val add2 x = T.succ(T.succ(x))
  end
```

then the new structure declaration

```
structure U = F(S)
```

is equivalent to having written

```
structure U =
  struct
    val y = x + 1
    val add2 x = succ(succ(x))
    val x = 5
    val succ x = x+1
  end
```

except that the signature of U does not include bindings for x and succ (i.e., they are hidden).

Although seemingly simple, the ML module facility has one very noteworthy feature: Structures are (at least partially) first class in that functor take them as arguments and return them as values. A more conservative design (such as adopted in Miranda and Haskell) might require all modules to be named, thus relegating them to second-class status. Of course, this first-class treatment has to be ended somewhere if type checking is to remain effective, and

in ML that is manifested in the fact that structures can be passed to functors *only* (e.g., they cannot be placed in lists), the signature declaration of a functor's argument is *mandatory* in the functor declaration, and functors themselves are not first class.

It is not clear whether this almost-first-class treatment of structures is worth the extra complexity, but the ML module facility is certainly the most sophisticated of those found in existing functional programming languages, and it is achieved with no loss of static type-checking ability, including the fact that modules may be compiled independently and later linked via functor application.

1.7 SASL, KRC, and Miranda

At the same time ML and FP were being developed, David Turner, first at the University of St. Andrews and later at the University of Kent, was busy working on another style of functional languages resulting in a series of three languages that characterize most faithfully the modern school of functional programming ideas. More than any other researcher, Turner [1981, 1982] argued eloquently for the value of lazy evaluation, higher order functions and the use of recursion equations as a syntactic sugaring for the lambda calculus. Turner's use of recurrence equations was consistent with Landin's argument 10 years earlier, as well as Burge's [1975] excellent treatise on recursive programming techniques and Burstall and Darlington's [1977] work on program transformation. But the prodigious use of higher order functions and lazy evaluation, especially the latter, was something new and was to become a hallmark of modern functional programming techniques.

In the development of SASL (St. Andrews Static Language) [Turner 1976], KRC (Kent Recursive Calculator) [Turner 1981], and Miranda[17] [Turner 1985], Turner concentrated on making things easier on the programmer, and thus he introduced various sorts of syntactic sugar. In particular, using SASL's syntax for equations gave programs a certain mathematical flavor, since equations were deemed applicable through the use of *guards* and Landin's notion of the off-side rule was revived. For example, this definition of the factorial function

$$\begin{aligned} \text{fac } n &= 1, && n{=}0 \\ &= n * \text{fac}(n{-}1), && n{>}0 \end{aligned}$$

looks a lot like the mathematical version

$$fac\ n = \begin{cases} 1 & \text{if } n = 0 \\ n * fac(n-1) & \text{if } n > 0 \end{cases}$$

(In Haskell this program would be written with slightly different syntax as

```
fac n | n==0 = 1
      | n>0  = n*fac(n-1)
```

More on equations and pattern matching may be found in Section 2.4.)

Another nice aspect of SASL's equational style is that it facilitates the use of higher order functions through *currying*. For example, if we define

$$\text{add } x\ y = x{+}y$$

then "add" 1 is a function that adds 1 to its argument.

KRC is an embellishment of SASL primarily through the addition of *ZF expressions* (which were intended to resemble Zemelo–Frankel set abstraction and whose syntax was originally suggested by John Darlington), as well as various other shorthands for lists (such as $[a\ ..\ b]$ to denote the list of integers from a to b, and $[a\ ..]$ to denote the infinite sequence starting with a). For example (using Haskell syntax),

$$[\ x{*}x \mid x \leftarrow [1\ ..\ 100],\ \text{odd}(x)\ ]$$

is the list of squares of the odd numbers from 1 to 100 and is similar to

$$\{x^2 \mid x \in \{1, 2, \ldots, 100\} \wedge odd(x)\}$$

except that the former is a list, the latter is a set. In fact Turner used the term *set abstraction* as synonymous with *ZF expression*, but in fact both terms are somewhat misleading since the expressions actually denote lists, not true sets. The more

[17] Miranda is one of the few (perhaps the only) functional languages to be marketed commercially.

popular current term is *list comprehension*,[18] which is what is used in the remainder of this paper. As an example of the power of list comprehensions, here is a concise and perspicuous definition of *quicksort*:

$$qs[\,] = [\,]$$
$$qs\,(x:xs) = qs\,[y \mid y \leftarrow xs, y<x\,] ++ [x] ++ qs[y \mid y \leftarrow xs, y>=x]$$

[++ is the infix append operator.]

Miranda is in turn an embellishment of KRC, primarily in its treatment of types: It is strongly typed, using a Hindley–Milner type system, and it allows user-defined concrete and abstract datatypes (both of these ideas were presumably borrowed from ML; see Sections 1.6 and 1.6.1). One interesting innovation in syntax in Miranda is its use of *sections* (first suggested by Richard Bird), which are a convenient way to convert partially applied infix operators into functional values. For example, the expressions $(+)$, $(x+)$, and $(+x)$ correspond to the functions f, g, and h, respectively, defined by

$$f\,x\,y = x+y$$
$$g\,y = x+y$$
$$h\,y = y+x$$

In part because of the presence of sections, Miranda does not provide syntax for lambda abstractions. (In contrast, the Haskell designers chose to have lambda abstractions and thus chose not to have sections.)

Turner was perhaps the foremost proponent of both higher-order functions and lazy evaluation, although the ideas originated elsewhere. Discussion of both of these topics is delayed until Sections 2.1 and 2.2, respectively, where they are discussed in a more general context.

1.8 Dataflow Languages

In the area of computer architecture, beginning predominantly with the work of Dennis' and Misuras [1974] the early 1970s there arose the notion of *dataflow*, a computer architecture organized solely around the data dependencies in a program, resulting in high degrees of parallelism. Since data dependencies were paramount and artificial sequentiality was objectionable, the languages designed to support such machines were essentially functional languages, although historically they have been called *dataflow languages*. In fact they do have a few distinguishing features, typically reflecting the idiosyncrasies of the dataflow architecture (just as imperative languages reflect the von Neumann architecture): They are typically first order (reflecting the difficulty in constructing closures in the dataflow model), strict (reflecting the data-driven mode of operation that was most popular and easiest to implement), and in certain cases do not even allow recursion (reflecting Dennis' original static dataflow design, rather than, for example, Arvind's dynamic tagged-token model [Arvind and Gostelow 1977; Arvind and Kathail 1981]). A good summary of work on dataflow machines, at least through 1981, can be found in Treleaven et al. [1982]; more recently, see Vegdahl [1989].

The two most important dataflow languages developed during this era were Dennis et al.'s Val [Ackerman and Dennis 1979; McGraw 1982], and Arvind and Gostelow's Id [1982]. More recently, Val has evolved into SISAL [McGraw et al. 1983] and Id into Id Nouveau [Nikhiel et al. 1986]. The former has retained much of the strict and first-order semantics of dataflow languages, whereas the latter has many of the features that characterize modern functional languages.

Keller's FGL [Keller et al. 1980] and Davis' DDN [Davis 1978] are also notable developments that accompanied a flurry of activity on dataflow machines at the University of Utah in the late 70's. Yet another interesting dataflow language is Ashcroft and Wadge's Lucid [Ashcroft and Wadge 1976a, 1976b; Wadge and Ashcroft 1985] [McGraw et al. 1983] whose distinguishing feature is the use of identifiers to represent streams of values (in a temporal, dataflow sense), thus allowing the expression of iteration in a rather concise manner. The authors also developed an algebra for reasoning about Lucid programs.

[18] A term popularized by Philip Wadler.

1.9 Others

In the late 1970s and early 1980s a surprising number of other modern functional languages appeared, most in the context of implementation efforts. These included Hope at Edinburgh University [Burstall et al. 1980], FEL at Utah [Keller 1982], Lazy ML (LML) at Chalmers [Augustsson 1984], Alfl at Yale [Hudak 1984], Ponder at Cambridge [Fairbairn 1985], Orwell at Oxford [Wadler and Miller 1988], Daisy at Indiana [Johnson N.d.] Twentel at the University of Twente [Kroeze 1987], and Tui at Victoria University [Boutel 1988].

Perhaps the most notable of these languages was Hope, designed and implemented by Rod Burstall, David MacQueen, and Ron Sannella at Edinburgh University [1980]. Their goal was "to produce a very simple programming language which encourages the production of clear and manipulable programs." Hope is strongly typed, allows polymorphism, but requires explicit type declarations as part of all function definitions (which also allowed a useful form of overloading). It has lazy lists but otherwise is strict. It also has a simple module facility. But perhaps the most significant aspect of Hope is its user-defined concrete datatypes and the ability to pattern match against them. ML, in fact, did not originally have these features; they were borrowed from Hope in the design of SML.

To quote Bird and Wadler [1988], this proliferation of functional languages was "a testament to the vitality of the subject," although by 1987 there existed so many functional languages that there truly was a Tower of Babel, and something had to be done. The funny thing was, the semantic underpinnings of these languages were fairly consistent, and thus the researchers in the field had very little trouble understanding each other's programs, so the motivation within the *research* community to standardize on a language was not high.

Nevertheless, in September 1987 a meeting was held at the FPCA Conference in Portland, Oregon, to discuss the problems that this proliferation of languages was creating. There was a strong consensus that the general use of modern, nonstrict functional languages was being hampered by the lack of a common language. Thus it was decided that a committee should be formed to design such a language, providing faster communication of new ideas, a stable foundation for real applications development, and a vehicle through which other people would be encouraged to learn and use functional languages. The result of that committee's effort was a purely functional programming language called Haskell [Hudak and Wadler 1988], named after Haskell B. Curry, and described in Section 1.10.

1.10 Haskell

Haskell is a general-purpose, purely functional programming language exhibiting many of the recent innovations in functional (as well as other) programming language research, including higher order functions, lazy evaluation, static polymorphic typing, user-defined datatypes, pattern matching, and list comprehensions. It is also a very complete language in that it has a module facility, a well-defined functional I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. In this sense Haskell represents both the culmination and solidification of many years of research on functional languages—the design was influenced by languages as old as Iswim and as new as Miranda.

Haskell also has several interesting new features; most notably, a systematic treatment of overloading, an orthogonal abstract datatype facility, a universal and purely functional I/O system, and, by analogy to list comprehensions, a notion of array comprehensions.

Haskell is not a small language. The decision to emphasize certain features such as pattern matching and user-defined datatypes and the desire for a complete and practical language that includes such things as I/O and modules necessitates a somewhat large design. The Haskell Report also provides a denotational semantics for both the static and dynamic behavior of the language; it is considerably more complex than

the simple semantics defined in Section 2.5 for the lambda calculus, but then again one wouldn't really want to program in as sparse a language as the lambda calculus.

Will Haskell become a standard? Will it succeed as a useful programming language? Only time will tell. As with any other language development, it is not only the quality of the design that counts but also the ready availability of good implementations and the backing from vendors, government agencies, and researchers alike. At this date it is too early to tell what role each of these factors will play.

I will end our historical development of functional languages here, without elaborating on the details of Haskell just yet. Those details will surface in significant ways in the next section, where the most important features of modern function languages are discussed, and in the following section, where more advanced ideas and active research areas are discussed.

2. DISTINGUISHING FEATURES OF MODERN FUNCTIONAL LANGUAGES

Recall that I chose to delay detailed discussion of four distinguishing features of modern functional languages—higher-order functions, lazy evaluation, data abstraction mechanisms, and equations/pattern matching. Now that we have completed our study of the historical development of functional languages, we can return to those features. Most of the discussion will center on how the features are manifested in Haskell, ML, and Miranda.

2.1 Higher Order Functions

If functions are treated as first-class values in a language—allowing them to be stored in data structures, passed as arguments, and returned as results—they are referred to as higher-order functions. I have not said too much about the use of higher order functions thus far, although they exist in most of the functional languages that I have discussed, including of course the lambda calculus. Their use has in fact been argued in many circles, including ones outside of functional programming, most notably the Scheme community [Abelson et al. 1985].

The main philosophical argument for higher-order functions is that functions are values just like any others, so why not give them the same first class status? But there are also compelling pragmatic reasons for wanting higher-order functions. Simply stated, the function is the primary abstraction mechanism over values; thus facilitating the use of functions increases the use of that kind of abstraction.

As an example of a higher-order function, consider the following:

$$\text{twice } f\ x = f\ (f\ x)$$

which takes its first argument, a function f, and applies it twice to its second argument, x. The syntax used here is important: twice as written is *curried*, meaning that when applied to one argument it returns a function that then takes one more argument, the second argument above. For example, the function add2.

```
add2 = twice succ
       where succ x = x+1
```

is a function that will add 2 to its argument. Making function application associate to the left facilitates this mechanism, since (twice succ x) is equivalent to ((twice succ) x), so everything works out just fine.

In modern functional languages functions can be created in several ways. One way is to name them using equations, as above; another way is to create them directly as *lambda abstractions*, thus rendering them nameless, as in the Haskell expression

```
\x → x+1
```

[in lambda calculus this would be written $\lambda x.x + 1$], which is the same as the successor function succ defined above. add2 can then be defined more succinctly as

```
add2 = twice (\x → x+1)
```

From a pragmatic viewpoint, we can understand the use of higher-order functions by analyzing the use of abstraction in general. As known from introductory programming, a function is an abstraction of values over some common behavior (an expression). Limiting the values over which the abstraction occurs to nonfunctions seems

unreasonable; lifting that restriction results in higher-order functions. Hughes makes a slightly different but equally compelling argument in Hughes [1984] where he emphasizes the importance of modularity in programming and argues convincingly that higher-order functions increase modularity by serving as a mechanism for glueing program fragments together. That glueing property comes not just from the ability to *compose* functions but also from the ability to abstract over functional behavior as described above.

As an example, suppose in the course of program construction we define a function to add together the elements of a list as follows:

```
sum [ ]    = 0
sum(x:xs) = add x (sum xs)
```

Then suppose we later define a function to multiply the elements of a list as follows:

```
prod [ ]    = 1
prod(x:xs)= mul x (prod xs)
```

But now we notice a repeating pattern and anticipate that we might see it again, so we ask ourselves if we can possibly abstract the common behavior. In fact, this is easy to do: We note that add/mul and 0/1 are the variable elements in the behavior, and thus we parameterize them; that is, we make them formal parameters, say *f* and init. Calling the new function fold, the equivalent of sum/prod will be "fold *f* init", and thus we arrive at

```
(fold f init) [ ]    = init
(fold f init)(x:xs) = f x ((fold f init) xs)
```

where the parentheses around "fold *f* init" are used only for emphasis, and are otherwise superfluous.

From this we can derive new definitions for sum and product:

```
sum  = fold add 0
prod = fold mul 1
```

Now that the fold abstraction has been made, many other useful functions can be defined, even something as seemingly unrelated as append:

```
append xs ys = fold (:) ys xs
```

[An infix operator may be passed as an argument by enclosing it in parentheses; thus (:) is equivalent to $\backslash x\ y \rightarrow x:y$.] This version of append simply replaces the [ ] at the end of the list *xs* with the list *ys*.

It is easy to verify that the new definitions are equivalent to the old using simple equational reasoning and induction. It is also important to note that in arriving at the main abstraction we did nothing out of the ordinary—we just apply classical data abstraction principles in as unrestricted a way as possible, which means allowing functions to be first-class citizens.

2.2 Nonstrict Semantics (Lazy Evaluation)

2.2.1 Fundamentals

The normal-order reduction rules of the lambda calculus are the most general in that they are guaranteed to produce a normal form if in fact one exists (see Section 1.1). In other words, they result in termination of the rewriting process most often, and the strategy lies at the heart of the Church–Rosser theorem. Furthermore, as argued earlier, normal-order reduction allows recursion to be emulated with the *Y* combinator, thus giving the lambda calculus the most powerful form of effective computability, captured in Church's Thesis.

Given all this, it is quite natural to consider using normal-order reduction as the computational basis for a programming language. Unfortunately, normal-order reduction, implemented naively, is hopelessly inefficient. To see why, consider this simple normal-order reduction sequence:

```
    (λx. (+ x x))(* 5 4)
⇒ (+ (* 5 4)(* 5 4))
⇒ (+ 20 (* 5 4))
⇒ (+ 20 20)
⇒ 40
```

Note that the multiplication (* 5 4) is done twice. In the general case, this could be an arbitrarily large computation, and it could potentially be repeated as many times as there are occurrences of the formal parameter (for this reason an analogy is often drawn between normal-order reduction and call-by-name parameter passing in Algol).

In practice this can happen quite often, reflecting the simple fact that results are often shared.

One solution to this problem is to resort to something other than normal-order reduction, such as applicative-order reduction, which for the above term yields the following reduction sequence:

$$\begin{aligned} & (\lambda x.\ (+\ x\ x))(*\ 5\ 4) \\ \Rightarrow\ & (\lambda x.\ (+\ x\ x))\ 20 \\ \Rightarrow\ & (+\ 20\ 20) \\ \Rightarrow\ & 40 \end{aligned}$$

Note that the argument is evaluated before the β-reduction is performed (similar to a call-by-value parameter-passing mechanism), and thus the normal form is reached in three steps instead of four, with no recomputation. The problem with this solution is that it requires the introduction of a special reduction rule to implement recursion (such as gained through the δ-rule for McCarthy's conditional), and furthermore there are examples for which it does more work than normal-order reduction. For example, consider

Applicative order	normal order
$(\lambda x.\ 1)(*\ 5\ 4)$	$(\lambda x.\ 1)(*\ 5\ 4)$
$\Rightarrow (\lambda x.\ 1)\ 20$	$\Rightarrow 1$
$\Rightarrow 1$	

or even worse (repeated from Section 1.1):

Applicative order

$$\begin{aligned} & (\lambda x.\ 1)((\lambda x.\ x\ x)(\lambda x.\ x\ x)) \\ \Rightarrow\ & (\lambda x.\ 1)((\lambda x.\ x\ x)(\lambda x.\ x\ x)) \\ \Rightarrow\ & \vdots \end{aligned}$$

Normal order

$$\begin{aligned} & (\lambda x.\ 1)((\lambda x.\ x\ x)(\lambda x.\ x\ x)) \\ \Rightarrow\ & 1 \end{aligned}$$

which in the applicative-order case does not terminate. Despite these problems, most of the early functional languages, including pure Lisp, FP, ML, Hope, and all of the dataflow languages used applicative-order semantics.[19] In addition to overcoming the efficiency problem of normal-order reduction, applicative-order reduction could be implemented with relative ease using the call-by-value compiler technology that had been developed for conventional imperative programming languages.

Nevertheless, the appeal of normal-order reduction cannot be ignored. Returning to lambda calculus basics, we can try to get to the root of the efficiency problem, which seems to be the following: Reduction in the lambda calculus is normally described as *string reduction*, which precludes any possibility of sharing. If instead we were to describe it as a *graph reduction* process, perhaps sharing could be achieved. This idea was first suggested by Wadsworth in his Ph.D. dissertation in 1971 [1971, Chapter 4], in which he outlined a graph-reduction strategy that used pointers to implement sharing. Using this strategy results in the following reduction sequence for the first example given earlier:

$$\begin{aligned} & (\lambda x.\ (+\ x\ x))(*\ 5\ 4) \\ \Rightarrow\ & (+\ \bullet\ \bullet) \\ & \quad\ \searrow\ \swarrow \\ & \quad\ \overbrace{(*\ 5\ 4)} \\ \Rightarrow\ & (+\ \bullet\ \bullet) \\ & \quad\ \searrow\ \swarrow \\ & \quad\ \underbrace{20} \\ \Rightarrow\ & 40 \end{aligned}$$

which takes the same number of steps as the applicative-order reduction sequence.

We will call an implementation of normal-order reduction in which recomputation is avoided *lazy evaluation* (another term often used is *call by need*). Its key feature is that arguments in function calls are evaluated at most once. It possesses the full power of normal-order reduction while

[19] Actually this is not quite true—most implementations of these languages use an applicative-order reduction strategy for the top-level redices *only*, thus yielding what is known as a weak head normal form. This strategy turns out to be easier to implement than complete applicative-order reduction and also permits certain versions of the Y combinator to be implemented without special rules. See Burg [1975] for an example of this using Landin's SECD machine.

being more efficient than applicative-order reduction in that "at most once" sometimes amounts to no computation at all.

Despite the appeal of lazy evaluation and this apparent solution to the efficiency problem, it took a good 10 years more for researchers to discover ways to implement it efficiently compared to conventional programming languages. The chief problem centered on the difficulty in implementing lazy graph-reduction mechanisms on conventional computers, which seem to be better suited to call-by-value strategies. Simulating the unevaluated portions of the graph in a call-by-value environment amounts to implementing *closures*, or "thunks" efficiently, which have some inherent, nontrivial costs [Bloss et al. 1981]. It is beyond the scope of this paper to discuss the details of these implementation concerns; see Peyton-Jones [1987] for an excellent summary.

Rather than live with conventional computers, we could alternatively build specialized graph-reduction or dataflow hardware, but so far this has not resulted in any practical, much less commercially available, machines. Nevertheless, this work is quite promising, and good summaries of work in this area can be found in articles by Treleaven et al. [1982] and Vegdahl [1984], both of which are reprinted in Thakkar [1987].

2.2.2 Expressiveness

Assuming that we can implement lazy evaluation efficiently (current practice is considered acceptably good), we should return to the question of why we want it in the first place. Previously we argued on philosophical grounds—it is the most general evaluation policy—but is lazy evaluation useful to programmers in practice? The answer is an emphatic yes, which I will show via a twofold argument.

First, *lazy evaluation frees a programmer from concerns about evaluation order.* The fact is, programmers are generally concerned about the efficiency of their programs, and thus they prefer not evaluating things that are not absolutely necessary. As a simple example, suppose we *may* need to know the greatest common divisor of b and c in some computation involving x. In a modern functional language we might write

```
f a x
where a = gcd b c
```

without worrying about a being evaluated needlessly—if in the computation of $f\ a\ x$ the value of a is needed, it will be computed; otherwise it will not. If we were to have written this program in Scheme, for example, we might try

```
(let ( (a (gcd b c)) )
  (f a x))
```

which will *always* evaluate a. Knowing that f doesn't always need that value and being concerned about efficiency, we may decide to rewrite this as

```
(let ( (a (delay (gcd b c))) )
  (f a x))
```

which requires modifying f so as to force its first argument. Alternatively, we could just write $(f\ b\ c\ x)$, which requires modifying f so as to compute the *gcd* internally. Both of these solutions are severe violations of modularity and arise out of the programmer's concern about evaluation order. Lazy evaluation eliminates that concern and preserves modularity.

The second argument for lazy evaluation is perhaps the one more often heard: *the ability to compute with unbounded "infinite" data structures.* The idea of lazily evaluating data structures was first proposed by Vuillemin [1974], but similar ideas were developed independently by Henderson and Morris [1976] and Friedman and Wise [1976]. In a later series of papers, Turner [1981, 1982] provided a strong argument for using lazy lists, especially when combined with list comprehensions (see Section 1.7) and higher order functions (see Section 2.1). Aside from Turner's elegant examples, Hughes [1984] presented an argument based on facilitating modularity in programming where, along with higher order functions, lazy evaluation is described as a way to "glue" pieces of programs together.

The primary power of lazily evaluated data structures comes from its use in separating data from control. The idea is that a programmer should be able to describe a specific data structure without worrying about how it gets evaluated. Thus, for example, we could describe the sequence of natural numbers by the following simple program:

```
nats = 0:map succ nats
```

or alternatively by

```
numsfrom n = n:numsfrom (n+1)
nats       = numsfrom 0
```

These are examples of "infinite lists", or streams, and in a language that did not support lazy evaluation would cause the program to diverge. With lazy evaluation these data structures are only evaluated as they are needed, on demand. For example, we could define a function that filters out only those elements satisfying a property p:

```
filter p (x:xs)
  = if (p x) then (x:rest) else rest
    where rest = filter p xs
```

in which case "filter p nats" could be written knowing that the degree of the list's computation will be determined by its context—that is, the consumer of the result. Thus filter has no operational control within it and can be combined with other functions in a modular way. For example, we could compose it with a function to square each element in a list:

```
map (\x→x*x) . filter p
```

[. is the infix composition operator.] This kind of modular result, in which data is separated from control, is one of the key features of lazy evaluation. Many more examples of this kind may be found in Hughes [1984].

2.3 Data Abstraction

Independently of the development of functional languages there has been considerable work on data abstraction in general and on strong typing, user-defined datatypes and type checking in particular. Some of this work has also taken on a theoretical flavor, not only in foundational mathematics where logicians have used types to resolve many famous paradoxes, but also in formal semantics where types aid our understanding of programming languages.

Fueling the theoretical work are two significant pragmatic advantages of using data abstraction and strong typing in one's programming methodology. First, data abstraction improves the modularity, security, and clarity of programs. Modularity is improved because we can abstract away from implementation (i.e., representation) details; security is improved because interface violations are automatically prohibited; and clarity is improved because data abstraction has an almost self-documenting flavor.

Second, strong static typing helps in debugging since we are guaranteed that if a program compiles successfully no error can occur at run time due to type violations. It also leads to more efficient implementations, since it allows us to eliminate the most run-time tag bits and type testing. Thus there is little performance penalty for using data abstraction techniques.

Of course, these issues are true for any programming language, and for that reason a thorough treatment of types and data abstraction is outside the scope of this survey; the reader may find an excellent summary in Cardelli and Wegner [1985]. The basic idea behind the Hindley–Milner type system was discussed in Section 1.6.1. I will concentrate in this section on how data abstraction is manifest in modern functional languages.

2.3.1 Concrete Datatypes

As mentioned earlier, there is a strong argument for wanting language features that facilitate data abstraction, whether or not the language is functional. In fact such mechanisms were first developed in the context of imperative languages such as Simula, Clu, and Euclid. It is only natural that they be included in functional languages. ML, as I mentioned, was the first functional language to do this, but many others soon followed suit.

In this section I will describe *concrete* (or *algebraic*) datatypes as well as *type synonyms*. I will use Haskell syntax, but the ideas are essentially identical (at least semantically) to those used in ML and Miranda.

New algebraic datatypes may be defined along with their constructors using data declarations, as in the following definitions of lists and trees:

```
data List a
        = Nil  |  Cons a (List a)
data Tree b
        = Empty | Node b (List (Tree b))
```

The identifiers List and Tree are called *type constructors*, and the identifiers *a* and *b* are called *type variables*, which are implicity universally quantified over the scope of the data declaration. The identifiers Nil, Cons, Empty, and Node are called *data constructors*, or just *constructors*, with Nil and Empty being *nullary constructors*. [Note that both type constructors and data constructors are capitalized, so that the latter can be used without confusion in pattern matching (as discussed in the next section) and to avoid confusion with type variables (such as *a* and *b* in the above example).]

List and Tree are called type constructors since they construct types from other types. For example, Tree Ints is the type of trees of integers. Reading from the data declaration for Tree, we see then that a tree of integers is either Empty or a Node containing an integer and a list of more trees of integers.

We can now see that the previously given type signature for map,

```
map :: (a → b) → [a] → [b]
```

is equivalent to

```
map :: (a → b) → (List a) → (List b)
```

That is, [. . .] in a type expression is just syntax for application of the type constructor List. Similarly, we can think of → as an infix type constructor that creates the type of all functions from its first argument (a type) to its second (also a type).

[A useful property to note is the consistent syntax used in Haskell for expressions and types. Specifically, if T_i is the type of expression or pattern e_i, then the expressions $\backslash e_1 \rightarrow e_2$, $[e_1]$, and (e_1,e_2) have the types $T_1 \rightarrow T_2$, $[T_1]$, and (T_1,T_2), respectively.]

Instances of these new types are built simply by using the constructors. Thus Empty is an empty tree, and Node 5 Empty is a very simple tree of integers with one element. The type of the instance is inferred via the same type inference mechanism that infers types of polymorphic functions, as described previously.

Defining new concrete datatypes is fairly common not only in functional languages but also in imperative languages, although the polymorphism offered by modern functional languages makes it all the more attractive.

Type synonyms are a way of creating new names for types, such as in the following:

```
type Intree     = Tree Ints
type Flattener = Intree → [Ints]
```

Note that Intree is used in the definition of Flattener.Type synonyms do not introduce new types (as data declarations do) but rather are a convenient way to introduce new names (i.e., synonyms) for existing types.

2.3.2 *Abstract Datatypes*

Another idea in data abstraction originating in imperative languages is the notion of an *abstract datatype* (ADT) in which the details of the implementation of a datatype are hidden from the users of that type, thus enhancing modularity and security. The traditional way to do this is exemplified by ML's ADT facility and emulated in Miranda. Although the Haskell designers chose a different approach to ADTs (described below), the following example of a queue ADT is written as if Haskell had ML's kind of ADTs, using the keyword abstype:

```
abstype Queue a = Q [a]
where first (Q as) = last as
      isempty (Q [ ]) = True
      isempty (Q as) = False
      ⋮
```

The main point is that the functions first, isempty, and so on, are visible in the scope

of the abstype declaration, but the constructor Q, including its type, is not. Thus a user of the ADT has no idea whether queues are implemented as lists (as shown here) or some other data structure. The advantage of this, of course, is that one is free to change the representation type without fear of breaking some other code that uses the ADT.

2.3.3 Haskell's Orthogonal Design

In Haskell a rather different approach was taken to ADTs. The observation was made that the main difference between a concrete and abstract datatype was that the latter had a certain degree of information hiding built into it. So instead of thinking of abstract datatypes and information hiding as going hand in hand, the two were made orthogonal components of the language. More specifically, concrete datatypes were made the only data abstraction mechanism, and to that an expose declaration was added to control information hiding, or visibility.

For example, to get the effect of the earlier definition of a queue, we would write

```
expose Queue, first, isempty
from data Queue a = Q [a]
     first (Q as) = last as
     isempty (Q [ ] = True
     isempty (Q as) = False
     ⋮
```

Since Q is not explicitly listed in the expose declaration, it becomes hidden from the user of the ADT.

The advantage of this approach to ADTs is more flexibility. For example, suppose we also wish to hide isempty or perhaps some auxiliary function defined in the nested scope. This is trivially done with the orthogonal design but is much harder with the ML design as described so far. Indeed, to alleviate this problem the ML designers provided an additional construct, a local declaration, with which one can hide local declarations. Another advantage of the orthogonal design is that the same mechanism can be used at the top level of a module to control visibility of the internals of the module to the external world. In other words, the expose mechanism is very general and can be nested. Haskell uses a conservative module system that relies on this capability.

A disadvantage of the orthogonal approach is that if the most typical ADT scenario only requires hiding the representative type, the user will have to think through the details in each case rather than having the hiding done automatically by using abstype.

2.4 Equations and Pattern Matching

One of the programming methodology attributes that is strongly encouraged in the modern school of functional programming is the use of equational reasoning in the design and construction of programs. The lack of side effects accounts for the primary ability to apply equational reasoning, but there are syntactic features that can facilitate it as well. Using equations as part of the syntax is the most obvious of these, but along with that goes *pattern matching* whereby one can write several equations when defining the same function, only one of which is presumably applicable in a given situation. Thus modern functional languages have tried to maximize the expressiveness of pattern matching.

At first blush, equations and pattern matching seem fairly intuitive and relatively innocuous. Indeed, we have already given many examples that use pattern matching without having said much about the details of how it works. But in fact pattern matching can have surprisingly subtle effects on the semantics of a language and thus should be carefully and precisely defined.

2.4.1 Pattern Matching Basics

Pattern matching should actually be viewed as the primitive behavior of a case expression, which has the general form

```
case e of
  pat1 → e1
  pat2 → e2
  ⋮
  patn → en
```

Intuitively, if the structure of e matches pati, then the result of the case expression

is ei. A set of equations of the form

```
f pat1 = e1
f pat2 = e2
  ⋮
f patn = en
```

can then be thought of as shorthand for

```
f = \x → case x of
      pat1 → e1
      pat2 → e2
      ⋮
      patn → en
```

Despite this translation, for convenience I will use the equational syntax in the remainder of this section.

The question to be asked first is just what the pattern-matching process consists of. For example, what exactly are we pattern matching against? One idea that seems reasonable is to allow one to pattern match against constants and data structures. Thus, fac can be defined by

```
fac  0 = 1
'    n = n*fac(n−1)
```

[The tick mark in the second equation is an abbreviation for fac.] But note that this relies on a top-to-bottom reading of the program, since (fac 0) actually matches both equations. We can remove this ambiguity by adding a guard (recall the discussion in Section 1.7), which in Haskell looks like the following:

```
fac 0         = 1
'   n | n>0   = n*fac(n−1)
```

As we have already demonstrated through several examples, it is also reasonable to pattern match against lists:

```
length [ ]     = 0
'      (x:xs)  = 1 + length xs
```

and for that matter any data structure, including user-defined ones:

```
data Tree2 a
  = Leaf a | Branch (Tree2 a) (Tree2 a)
fringe (Leaf x)   = [x]
'      (Branch left right)
                  = fringe left ++ fringe right
```

where ++ is the infix append operator.

Another possibility that seems desirable is the ability to repeat formal parameters on the left-hand side to denote that arguments in those positions must have the same value, as in the second line of the following definition:

```
member x [ ]      = False
'      x (x:xs)   = True
'      x (y:xs)   = member x xs
```

[This is not legal Haskell syntax, since such repetition of identifiers is not allowed.] Care must, however, be taken with this approach since in something like

```
alleq [x, x, x] = True
'     y         = False
```

it is not clear in what order the elements of the list are to be evaluated. For example, if they are evaluated left to right then "alleq [1, 2, bot]", where bot is any nonterminating computation, will return False, whereas with a right to left order the program will diverge. One solution that would at least guarantee consistency in this approach is to insist that all three positions are evaluated so that the program diverges if any of the arguments diverge.

In general the problem of what gets evaluated, and when, is perhaps the most subtle aspect of reasoning about pattern matching and suggests that the pattern-matching algorithm be fairly simple so as not to mislead the user. Thus in Haskell the above repetition of identifiers is disallowed—equations must be *linear*—but some functional languages allow it (e.g., Miranda and Alfl [Hudak 1984]).

A particularly subtle version of this problem is captured in the following example. Consider these definitions:

```
data Silly a = Foo a | Other
bar (Foo x) = 0
'   Other   = 1
```

Then a call "bar bot" will diverge, since bar must be strict in order that it can distinguish between the two kinds of arguments that it might receive. But now consider this small modification:

```
data Silly a = Foo a
bar (Foo x) = 0
```

Now a call bar bot seems like it should return 0, since bar need not be strict—it can only receive one kind of argument and thus does not need to examine it unless it is needed in computing the result, which in this case it does not. In Haskell a mechanism is provided so that *either* semantics may be specified.

Two useful discussions on the subject of pattern matching can be found in Augustsson [1985] and Wadler [1987a].

2.4.2 Connecting Equations

Let us now turn to the more global issue of how the individual equations are connected together as a group. As mentioned earlier, one simple way to do this is give the equations a top-to-bottom priority, so that in

```
fac 0 = 1
'   n = n * fac(n-1)
```

the second equation is tried only after the first one has failed. This is the solution adopted in many functional languages, including Haskell and Miranda.

An alternative method is to insist that the equations be disjoint, thus rendering the order irrelevant. One significant motivation for this is the desire to reason about the applicability of an equation independently of the others, thus facilitating equational reasoning. The question is, How can one guarantee disjointness? For equations without guards, the disjointness property can be determined statically; that is, by just examining the patterns. Unfortunately, when unrestricted guards are allowed, the problem becomes undecidable, since it amounts to determining the equality of arbitrary recursive predicates. This in turn can be solved by resolving the guard disjointness at run time. On the other hand, this solution ensures correctness only for values actually encountered at run time, and thus the programmer might apply equational reasoning erroneously to as yet unencountered values.

The two ideas could also be combined by providing two different syntaxes for joining equations. For example, using the hypothetical keyword else (not valid in Haskell):

```
sameShallowStructure [a]   [c]   = True
'                    [a,b] [c,d] = True
else
'                    x     y     = False
```

The first two equations would be combined using a disjoint semantics; together they would then be combined with the third using a top-to-bottom semantics. Thus the third equation acts as an "otherwise" clause in the case that the first two fail. A design of this sort was considered for Haskell early on, but the complexities of disjointness, especially in the context of guards, were considered too great and the design was eventually rejected.

2.4.3 Argument Order

In the same sense that it is desirable to have the order of equations be irrelevant, it is desirable to have the order of arguments be irrelevant. In exploring this possibility, consider first the functions f and g defined by

```
f 1 1 = 1
f 2 x = 2
g 1 1 = 1
g x 2 = 2
```

which differ only in the order of their arguments. Now consider to what "f 2 bot" should evaluate. Clearly the equations for f are disjoint, clearly the expression matchs only the second equation, and since we want a nonstrict language it seems the answer should clearly be 2. For a compiler to achieve this it must always evaluate the first argument to f first.

Now consider the expression "g bot 2"—by the same argument given above the result should also be 2, but now the compiler must be sure always to evaluate the second argument to g first. Can a compiler always determine the correct order in which to evaluate its arguments?

To help answer that quation, first consider this intriguing example (due to Berry [1978]):

```
f 0 1 x = 1
f 1 x 0 = 2
f x 0 1 = 3
```

Clearly these equations are disjoint. So what is the value of "f 0 1 bot"? The desired answer is 1. And what is the value of "f 1 bot 0"? The desired answer is 2. And what is the value of "f bot 0 1"? The desired answer is 3. But now the most difficult question is In what order should the arguments be evaluated? If we evaluate the third one first, then the answer to the first question cannot be 1. If we evaluate the second one first, then the answer to the second question cannot be 2. If we evaluate the first one first, then the answer to the third question cannot be 3. In fact there is *no* sequential order that will allow us to answer these three questions the way we would like—some kind of parallel evaluation is required.

This subtle problem is solvable in several ways, but they all require some kind of compromise—insisting on a parallel (or pseudoparallel) implementation, rejecting certain seemingly valid programs, making equations more strict than one would like, or giving up on the independence of the order of evaluation of arguments. In Haskell the last solution was chosen—perform a left-to-right evaluation of the arguments—because it presented the simplest semantics to the user, which was judged to be more important than making the order of arguments irrelevant. Another way to explain this is to think of equations as syntax for nested lambda expressions, in which case one might not expect symmetry with respect to the arguments anyway.

2.5 Formal Semantics

Simultaneously with work on functional languages Scott, Strachey, and others were busy establishing the foundations of denotational semantics, now the most widely used tool for describing the formal semantics of programming languages. There was a close connection between this work and functional languages primarily because the lambda calculus served as one of the simplest programming languages with enough useful properties to make its formal semantics interesting. In particular, the lambda calculus had a notion of *self-application*, which implies that certain domains had to contain their own function spaces. That is, it required a domain D that was a solution to the following domain equation:

$$D = D \rightarrow D$$

At first this seems impossible—surely there are more functions from D into D than there are elements in D—but Scott [1970] was able to show that indeed such domains existed, as long as one was willing to restrict the allowable functions in certain (quite reasonable) ways and by treating = as an isomorphism rather than an equality. Scott's work served as the mathematical foundation for Strachey's work [Milne and Stracheg 1976] on the denotational semantics of programming languages; see [Stoy 1979] and [Schmidt 1985] for thorough treatments.

Denotational semantics and functional programming have close connections, and the functional programming community emphasizes the importance of formal semantics in general. For completeness and to show how simple the denotational semantics of a functional language can be, we give the semantics of the recursive lambda calculus with constants defined in Section 2.3.

$$\begin{array}{lll} Bas & = Int + Bool + \cdots & \text{Basic values} \\ D & = Bas + (D \rightarrow D) & \text{Denotable values} \\ Env & = Id \rightarrow D & \text{Environments} \end{array}$$

$$\mathscr{E}: Exp \rightarrow Env \rightarrow D$$
$$\mathscr{K}: Con \rightarrow D$$

$$\mathscr{E}[\![x]\!]env = env[\![x]\!]$$
$$\mathscr{E}[\![c]\!]env = \mathscr{K}[\![c]\!]$$
$$\mathscr{E}[\![e_1 e_2]\!]env = (\mathscr{E}[\![e_1]\!]env)(\mathscr{E}[\![e_2]\!]env)$$
$$\mathscr{E}[\![\lambda x.e]\!]env = \lambda v.\mathscr{E}[\![e]\!]env[v/x]$$
$$\mathscr{E}[\![e \text{ where } x_1 = e_1; \ldots; x_n = e_n]\!]env = \mathscr{E}[\![e]\!]env'$$

where

$$env' = fix\ \lambda env'.env[(\mathscr{E}[\![e_1]\!]env')/x_1, \ldots, (\mathscr{E}[\![e_n]\!]env')/x_n]$$

This semantics is relatively simple, but in moving to a more complex language such as Miranda or Haskell the semantics can become significantly more complex due to the many syntactic features that make the languages convenient to use. In addition,

one must state precisely the static semantics as well, including type checking and pattern-matching usage. This complexity is managed in the Haskell Report by first translating Haskell into a *kernel* which is only slightly more complex than the above.

3. ADVANCED FEATURES AND ACTIVE RESEARCH AREAS

Some of the most recent ideas in functional language design are new enough that they should be regarded as on-going research. Nevertheless, many of them are sound enough to have been included in current language designs. In this section we will explore a variety of such ideas, beginning with some of the innovative ideas in the Haskell design. Some of the topics have a theoretical flavor, such as extensions of the Hindley–Milner type system; some have a pragmatic flavor, such as expressing nondeterminism, efficient arrays, and I/O; and some involve the testing of new application areas, such as parallel and distributed computation. All in all, studying these topics should provide insight into the goals of functional programming as well as some of the problems in achieving those goals.

3.1 Overloading

The kind of polymorphism exhibited by the Hindley–Milner type system is what Strachey called *parametric polymorphism* to distinguish it from another kind that he called *ad hoc polymorphism* or *overloading*. The two can be distinguished in the following way: A function with parametric polymorphism does not care what type certain of its arguments have and thus it behaves identically regardless of the type. In contrast, a function with ad hoc polymorphism does care and in fact may behave differently for different types. Stated another way, ad hoc polymorphism is really just a syntactic device for overloading a particular function name or symbol with more than one meaning.

For example, the function map defined earlier exhibits parametric polymorphism and has typing

$$\text{map} :: (a \to b) \to [a] \to [b]$$

Regardless of the kind of list given to map it behaves in the same way. In contrast, consider the function +, which we normally wish to behave differently for integer and floating point numbers and not at all (i.e., be a static error) for nonnumeric arguments. Another common example is the function == (equality), which certainly behaves differently when comparing the equality of two numbers versus, say, two lists.

Ad hoc polymorphism is normally (and I suppose appropriately) treated in an ad hoc manner. Worse, there is no accepted convention for doing this; indeed ML, Miranda, and Hope all do it differently. Recently, however, a uniform treatment for handling overloading was discovered independently by Kaes [1988] (in trying to generalize ML's ad hoc equality types) and Wadler and Blott [1989] (as part of the process of defining Haskell). Below I will describe the solution as adopted in Haskell; details may be found in Wadler and Blott [1989].

The basic idea is to introduce a notion of *type classes* that capture a collection of overloaded operators in a consistent way. A *class declaration* is used to introduce a new type class and the overloaded operators that must be supported by any type that is an instance of that class. An *instance declaration* declares that a certain type is an instance of a certain class, and thus included in the declaration are the definitions of the overloaded operators instantiated on the named type.

For example, say that we wish to overload + and negate on types Int and Float. To do so, we introduce a new type class called Num:

$$\begin{aligned}&\text{class Num } a \text{ where}\\&\quad (+) \quad :: a \to a \to a\\&\quad \text{negate} :: a \to a\end{aligned}$$

This declaration may be read "a type a belongs to the class Num if there are (overloaded) functions + and negate, of the appropriate types, defined on it."

We may then declare Int and Float to be instances of this class, as follows:

$$\begin{aligned}&\text{instance Num Int where}\\&\quad x + y \quad = \text{addInt } x\ y\\&\quad \text{negate } x = \text{negateInt } x\end{aligned}$$

```
instance Num Float where
  x + y    = addFloat x y
  negate x = negateFloat x
```

[note how infix operators are defined; Haskell's lexical syntax prevents ambiguities] where addInt, negateInt, addFloat, and negateFloat are assumed in this case to be predefined functions but in general could be any user-defined function. The first declaration above may be read "Int is an instance of the class Num as witnessed by these definitions of + and negate."

Using type classes we can thus treat overloading in a consistent, arguably elegant, way. Another nice feature is that type classes naturally support a notion of inheritance. For example, we may define a class Eq by

```
class Eq a where
  (==) :: a → a → Bool
```

Given this class, we would certainly expect all members of the class Num, say, to have == defined on them. Thus the class declaration for Num could be changed to

```
class Eq a ⇒ Num a where
  (+)    :: a → a → a
  negate :: a → a
```

which can be read as "only members of the class Eq may be members of the class Num, and a type a belongs to the class Num if . . . ⟨as before⟩." Given this class declaration, instance declarations for Num must include a definition of == as in

```
instance Num Int where
  x + y    = addInt x y
  negate x = negateInt x
  x == y   = eqInt x y
```

The Haskell Report uses this inheritance mechanism to define a very rich hierarchical numeric structure that reflects fairly well a mathematician's view of numbers.

The traditional Hindley–Milner type system is extended in Haskell to include type classes. The resulting type system is able to verify that the overloaded operators do have the appropriate type. It is however, possible (but not likely) for ambiguous situations to arise, which in Haskell result in type error but can be reconciled explicitly by the user (see Hudak and Wadler [1988] for the details).

3.2 Purely Functional Yet Universal I/O

To many the notion of I/O conjures an image of state, side effects, and sequencing. Is there any hope at achieving purely functional yet universal and of course efficient I/O? Suprisingly, the answer is yes. Perhaps even more surprising is that over the years there have emerged not one but two seemingly very different solutions:

- The *lazy stream model*, in which temporal events are modeled as lists, whose lazy semantics mimics the demand-driven behavior of processes.
- The *continuation* model in which temporality is modeled via explicit continuations.

Although papers have been written advoacting both solutions, and indeed they are very different in style, the two solutions turn out to be exactly equivalent in terms of expressiveness; in fact, there is an almost trivial translation from one to the other. The Haskell I/O system takes advantage of this fact and provides a unified framework that supports both styles. The specific I/O operations available in each style are identical—what differs is the way they are expressed—and thus programs in either style may be combined with a well-defined semantics. In addition, although certain of the primitives rely on nondeterministic behavior in the operating system, referential transparency is still retained internal to a Haskell program.

In this section the two styles will be described as they appear in Haskell, together with the translation of one in terms of the other. Details of the actual Haskell design may be found in Hudak and Wadler [1988], and a good discussion of the tradeoffs between the styles, including examples, may be found in Hudak and Sundaresh [1988]. ML, by the way, uses an imperative, referentially opaque, form of I/O (perhaps not surprising given the presence of references); Miranda uses a rather restricted form of the stream model; and Hope uses the continuation model but with a strict (i.e., call-by-value) semantics.

To begin, we can paint an appealing functional view of a collection of programs executing within an operating system (OS)

as shown in Figure 1. With this view programs are assumed to communicate with the OS via messages—programs issue requests to the OS and receive responses from the OS.

Ignoring for now the OS itself as well as the merge and split operations, a program can be seen as a function from a stream (i.e., list) of responses to a stream of requests. Although the above picture is quite intuitive, this latter description may seem counterintuitive—how can a program receive a list of responses before it has generated any requests? But remember that we are using a *lazy* (i.e., nonstrict) language, and thus the program is not obliged to examine any of the responses before it issues its first request. This application of lazy evaluation is in fact a very common style of programming in functional languages.

Thus a Haskell program engaged in I/O is required to have type Behavior, where

```
type Behavior = [Response] → [Request]
```

(Recall from Section 2.3 that [Response] is the type consisting of lists of values of type Response.) The main operational idea is that the nth response is the reply of the operating system to the nth request.

For simplicity, we will assume that there are only two kinds of requests and three kinds of responses, as defined below:

```
data Request
  = ReadFile' Name
  | WriteFile' Name Contents

data Response
  = Success | Return Contents
            | Failure ErrorMsg
type Name     = String
type Contents = String
type ErrorMsg= String
```

[This is a subset of the requests available in Haskell.]

As an example, given this request list,

```
[ . . . , WriteFile fname s1, Readfile fname,
  . . . ]
```

and the corresponding response list,

```
[ . . . , Success, Return s2, . . . ]
```

then $s1$ == $s2$, unless there were some intervening external effect.

In contrast, the continuation model is normally characterized by a set of *transactions*. Each transaction typically takes a success continuation and a failure continuation as arguments. These continuations in turn are simply functions that generate more transactions. For example,

```
data Transaction
  = ReadFile'   Name FailCont RetCont
  | WriteFile'  Name Contents
                FailCont SuccCont
type FailCont
  = ErrorMsg → Transaction
type RetCont
  = Contents → Transaction
Type SuccCont Transaction
```

[In Haskell the transactions are actually provided as functions rather than constructors; see below.] The special transaction Done represents program termination. These declarations should be compared with those for the stream model given earlier.

Returning to the simple example given earlier, the request and response list are no longer separate entities since their effect is interwoven into the continuation structure, yielding something like this:

```
WriteFile' fname s1 exit
  (ReadFile' fname exit
    (\s2 → . . .) )
where exit errmsg = Done
```

in which case, as before, we would expect $s1$ == $s2$ in the absence of external effects. This is essentially the way I/O is handled in Hope.

Although these two styles seem very different, there is a simple translation of the continuation model into the stream model. In Haskell, instead of defining the new datatype Transaction, a set of functions is defined that accomplishes the same task but that is really stream transformers in

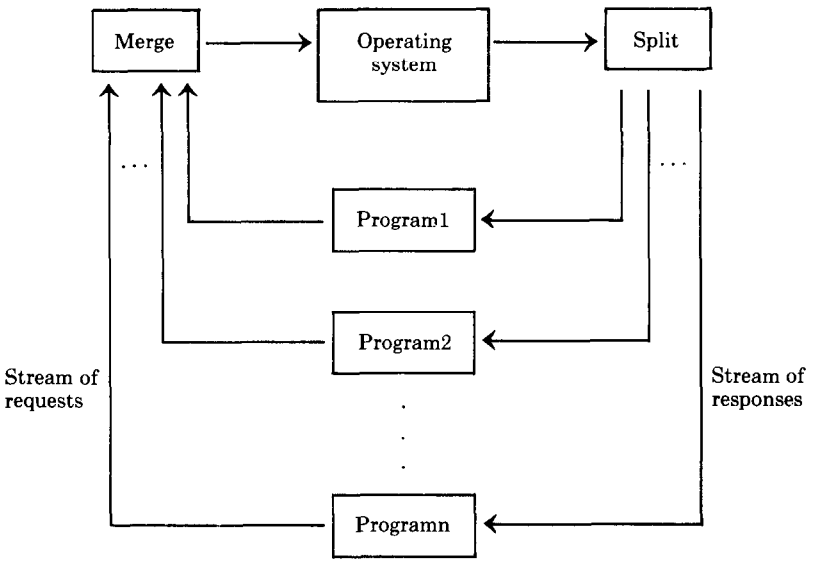

Figure 1. Functional I/O.

the request/response style. In other words, the type Transaction should be precisely the type Behavior and should not be a new datatype at all. Thus we arrive at

```
readFile  :: Name →                    FailCont → FailCont → RetCont → Behavior
writeFile :: Name → Contents → FailCont → SuccCont                   → Behavior
done      ::                                                           Behavior
type FailCont  = ErrorMsg → Behavior
type RetCont   = Contents → Behavior
type SuccCont =             Behavior
readFile name fail succ resps =
  (ReadFile name) : case (head resps) of
                      Return contents → succ contents (tail resps)
                      Failure msg     → fail msg (tail resps)
writeFile name contents fail succ resps =
  (WriteFile name contents) : case (head resps) of
                                Success     → succ (tail resps)
                                Failure msg → fail msg (tail resps)

done resps = [ ]
```

This pleasing and very efficient translation allows us to write Haskell programs in either style and mix them freely.

The complete design, of course, includes a fairly rich set of primitive requests besides ReadFile and WriteFile, including a set of requests for communicating through channels, which include things such as standard input and output. Because the merge itself is implemented in the operating system, referential transparency is retained within a Haskell program.

Another useful aspect of the Haskell design is that it includes a partial (but rigorous) specification of the behavior of the OS itself. For example, it introduces the notion of an *agent* that consumes data on output channels and produces data on input channels. The user is then modeled as an agent that consumes standard output and

produces standard input. This particular agent is required to be strict in the standard output, corresponding to the notion that the user reads the terminal display before typing at the keyboard. No other language design that I am aware of has gone this far in specifying the I/O system with this degree of precision; it is usally left implicit in the specification. It is particularly important in this context however, because the proper semantics relies critically on how the OS consumes and produces the request and response lists.

To conclude this section I will show two Haskell programs that prompt the user for the name of a file, read and echo the file name, and then look up and display the contents of the file on standard output. The first version uses the stream model, the second the continuation model.
[The operator !! is the list selection operator; thus *xs* !! *n* is the *n*th element in the list *xs*.]

```
main resps =
  [ AppendChannel "stdout" "please type a filename\CR\",
    if (resps!!1 == Success) then (ReadChannel "stdin"),
    AppendChannel "stdout" fname,
    if (resps!!3 == Success) then (ReadFile fname),
    AppendChannel "stdout" (case resps !! 4 of
                              Failure msg          → "can't open"
                              Return file_contents → file_contents)
  ] where fname = case resps !! 2 of
                    Return user_input → get_line user_input
```

```
main = appendChannel "stdout" "please type a filename\CR\" exit
       (readChannel "stdin" exit (\user_input →
         appendChannel "stdout" fname exit
       (readFile fname (\msg → appendChannel "stdout" "can't open" exit done)
         (\contents →
         appendChannel "stdout" contents exit done))
         where fname = get_line user_input))
exit msg = done
```

3.3 Arrays

As it turns out, arrays can be expressed rather nicely in a functional language, and in fact all of the arguments about mathematical elegance fall in line when using arrays. This is especially true of program development in scientific computation, where textbook matrix algebra can often be translated almost verbatim into a functional language. The main philosophy is to treat the entire array as a single entity defined declaratively rather than as a place holder of values that is updated incrementally. This, in fact, is the basis of the APL philosophy (see Section 1.4), and some researchers have concentrated on combining functional programming ideas with those from APL [Tu 1986; Tu and Perlis 1986]. The reader may find good general discussions of arrays in functional programming languages in Wadler [1986], Hudak [1986a], and Wise [1987]. In the remainder of this section I will describe Haskell's arrays, which originated from some ideas in Id Nouveau [Nikhil et al. 1986].

Haskell has a family of multidimensional nonstrict immutable arrays whose special interaction with list comprehensions provides a convenient array comprehension syntax for defining arrays monolithically. As an example, here is how to define a vector of squares of the integers from 1 to *n*:

$$a = \text{array } (1, n)\ [\ (i, i*i)\ |\ i \leftarrow [1\,..\,n]\ ]$$

The first argument to array is a tuple of *bounds*, and thus this array has size *n* and is indexed from 1 to *n*. The second

argument is a list of index/value pairs and is written here as a conventional list comprehension. The ith element of an array a is written $a\ !\ i$, and thus in the above case we have that $a\ !\ i = i*i$.

There are several useful semantic properties of Haskell's arrays. First, they can be recursive—here is an example of defining the first n numbers in the Fibonacci sequence:

```
fib = array (0, n)
( [ (0, 1), (1, 1) ] ++
  [ (i, fib!(i−1)+fib! (i−2)) | i ← [2 .. n] ] )
```

This example demonstrates how we can use an array as a cache, which in this case turns an exponentially poor algorithm into an efficient linear one.

Another important property is that array comprehensions are constructed lazily, and thus the order of the elements in the list is completely irrelevant. For example, we can construct an m-by-n matrix using a wavefront recurrence, where the north and west borders are 1 and each other element is the sum of its north, northwest, and west neighbors, as follows:

```
a = array ((1,m), (1,n))
  ( [ ((1,1),1) ]                          ++
    [ ((i,1),1) | i ← [2 .. m] ] ++
    [ ((1,j),1) | j ← [2 .. n] ] ++
    [ ((i,j), a!(i−1,j) + a!(i,j−1)
        + a!(i−1,j−1))
        | i ← [2 .. m], j ← [2 .. n] ] )
```

The elements in this result can be accessed in any order—the demand-driven effect of lazy evaluation will cause the necessary elements to be evaluated in an order constrained only by data dependencies. It is this property that makes array comprehensions so useful in scientific computation, where recurrence equations express the same kind of data dependencies. In implementing such recurrences in FORTRAN we must be sure that the elements are evaluated in an order consistent with the dependencies—lazy evaluation accomplishes that for us.

On the other hand, although elegant, array comprehensions may be difficult to implement efficiently. There are two main difficulties: the standard problem of overcoming the inefficiencies of lazy evaluation and the problem of avoiding the construction of the many intermediate lists that the second argument to array seems to need. A discussion of ways to overcome these problems is found in Anderson and Hudak [1989]. Alternative designs for functional arrays and their implementations may be found in Aasa et al. [1987], Holmstrom [1983], Hughes [1985a], and Wise [1987].

Another problem is that array comprehensions are not quite expressive enough to capture all behaviors. The most conspicuous example of this is the case in which an array is being used as an accumulator, say in building a histogram, and thus one actually wants an incremental update effect. Thus in Haskell a function called accumArray is provided to capture this kind of behavior in a way consistent with the monolithic nature of array comprehensions (similar ideas are found in Steele et al. [1986] and Wadler [1986]). It is not, however, clear that this is the most general solution to the problem. An alternative approach is to define an incremental update operator on arrays, but then even nastier efficiency problems arise, since (conceptually at least) the updates involve copying. Work on detecting when it is safe to implement such updates destructively has resulted in at least one efficient implementation [Bloss 1988; Bloss and Hudak N.d.; Hudak and Bloss 1985], although the analysis itself is costly.

Nevertheless, array comprehensions have the potential for being very useful, and many interesting applications have already been programmed using them (see Hudak and Anderson [1988] for some examples). It is hoped that future research will lead to solutions to the remaining problems.

3.4 Views

Pattern matching (see Section 2.4) is very useful in writing compact and readable programs. Unfortunately, knowledge of the concrete representation of an object is necessary before pattern matching can be invoked, which seems to be at odds with the notion of an abstract datatype. To reconcile this conflict Wadler [1987] introduced a notion of *views*.

A view declaration introduces a new algebraic datatype, just like a data declaration, but in addition establishes an isomorphism between the values of this new type and a subset of the values of an existing algebraic datatype. For example,

```
data Complex = Rectangular Float Float
view Complex = Polar       Float Float
  where toView (Rectangular x y)
    = Polar (sqrt (x**2+y**2))
            (arctan (y/x))
  fromView (Polar r t)
    = Rectangular (r*(cos t))
                  (r*(sin t))
```

[Views are not part of Haskell, but as with abstract datatypes we will use Haskell syntax, here extended with the keyword view.] Given the datatype Complex, we can read the view declaration as, "One view of Complex contains only Polar values; to translate from a Rectangular to a Polar value, use the function toView; to translate the other way, use fromView."

Having declared this view, we can now use pattern matching using either the Rectangular constructor or the Polar constructor; similarly, new objects of type Complex can be built with either the Rectangular or Polar constructors. They have precisely the same status, and the coercions are done automatically. For example,

```
rotate (Polar r t) angle = Polar r (t+angle)
```

As the example stands, objects of type Complex are concretely represented with the Rectangular constructor, but this decision could be reversed by making Polar the concrete constructor and Rectangular the view, without altering any of the functions that manipulate objects of type Complex.

Whereas traditionally abstract data types are regarded as hiding the representation, with views we can reveal as many representations (zero, one, or more) as are required.

As a final example, consider this definition of Peano's view of the natural number subset of integers:

```
view Integer = Zero | Succ Integer
  where fromView Zero = 0
        ' (Succ n) | n>=0   = n + 1
        toView 0 = Zero
        ' n | n>0  = Succ (n-1)
```

With this view, 7 is viewed as equivalent to

```
Succ (Succ (Succ (Succ (Succ (Succ (Succ
     Zero))))))
```

Note that fromView defines a mapping of any finite element of Peano into an integer, and toView defines the inverse mapping. Given this view, we can write definitions such as

```
fac Zero    = 1
' (Succ n) = (Succ n) * (fac n)
```

which is very useful from an equational reasoning standpoint, since it allows us to use an abstract representation of integers without incurring any performance overhead—the view declarations provide enough information to map all of the abstractions back into concrete implementations at compile time.

On the other hand, perhaps the most displeasing aspect of views is that an implicit coercion is taking place, which may be confusing to the user. For example, in

```
case (Foo a b) of
  Foo x y → exp
```

we cannot be sure in exp that $a==x$ and $b==y$. Although views were considered in an initial version of Haskell, they were eventually discarded, in a large part because of this problem.

3.5 Parallel Functional Programming

An often-heralded advantage of functional languages is that parallelism in a functional program is implicit; it is manifested solely through data dependencies and the semantics of primitive operators. This is in contrast to more conventional languages, where explicit constructs are typically used to invoke, synchronize, and in general coordinate the concurrent activities. In fact, as discussed earlier, many functional languages were developed simultaneously with work on highly parallel dataflow and reduction machines, and such research continues today.

In most of this work, parallelism in a functional program is detected by the system and allocated to processors automatically. Although in certain constrained

classes of functional languages the mapping of process to processor can be determined optimally [Chen 1986; Delosme and Ipsen 1985], in the general case the optimal strategy is undecidable, so heuristics such as load balancing are often used instead.

But what if a programmer knows a good (perhaps optimal) mapping strategy for a program executing on a particular machine, but the compiler is not smart enough to determine it? And even if the compiler is smart enough, how does one reason about such behavior? We could argue that the programmer should not be concerned about such details, but that is a difficult argument to make to someone whose job is precisely to invent such algorithms.

To meet these needs, various researchers have designed extensions to functional languages, resulting in what I like to call *parafunctional programming languages.* The extensions amount to a metalanguage (e.g., annotations) to express the desired behavior. Examples include annotations to control evaluation order [Burton 1984; Darlington and While 1987; Sridharan 1985], prioritize tasks, and map processes to processors [Hudak 1986c; Hudak and Smith 1986; Keller and Lindstrom 1985]. Similar work has taken place in the Prolog community [Shapiro 1984]. In addition, research has resulted in formal operational semantics for such extensions [Hudak 1986b; Hudak and Anderson 1987]. In the remainder of this section one kind of parafunctional behavior will be demonstrated—that of mapping program to machine (based on the work in Hudak [1986c] and Hudak and Smith [1986]).

The fundamental idea behind process-to-processor mapping is quite simple. Consider the expression e1+e2. The strict semantics of + allows the subexpressions e1 and e2 to be executed in parallel—this is an example of what is meant by saying that the parallelism in a functional program is implicit. But suppose now that we wish to express precisely *where* (i.e., on which processor) the subexpressions are to be evaluated; we may do so quite simply by annotating the subexpressions which appropriate mapping information. An expression annotated in this way is called a *mapped expression*, which has the following form:

```
exp on proc
```

[on is a hypothetical keyword, and is not valid Haskell] which intuitively declares that exp is to be computed on the processor identified by proc. The expression exp is the body of the mapped expression and represents the value to which the overall expression will evaluate (and thus can be any of expression, including another mapped expression). The expression proc must evaluate to a processor id. Without loss of generality the processor ids, or pids, are assumed to be integers, and there is some predefined mapping from those integers to the physical processors they denote.

Returning now to the example, we can annotate the expression (e1+e2) as follows:

```
(e1 on 0) + (e2 on 1)
```

where 0 and 1 are processor ids. Of course, this static mapping is not very interesting. It would be nice, for example, if we were able to refer to a processor relative to the currently executing one. We can do this through the use of the reserved identifier self, which when evaluated returns the pid of the currently executing processor. Using self we can now be more creative. For example, suppose we have a ring of n processors that are numbered consecutively; we may then rewrite the above expression as

```
(e1 on left self) + (e2 on right self)
    where left pid = mod (pid−1) n
          right pid = mod (pid+1) n
```

[mod x y computes x modulo y.], which denotes the computation of the two subexpressions in parallel on the two neighboring processors, with the sum being computed on self.

To see that it is desirable to bind self dynamically, consider that one may wish successive invocations of a recursive call to be executed on different processors—this is not easily expressed with lexically bound annotations. For example, consider the following list-of-factorials program, again using a ring of processors:

```
(map fac [2,3,4]) on 0
where map f []     = []
      ,     f (x:xs) = f x : ((map f xs) on
                              (right self))
```

Note that the recursive call to map is mapped onto the processor to the right of the current one, and thus the elements 2, 6, and 24 in the result list are computed on processors 0, 1, and 2, respectively.

Parafunctional programming languages have been shown to be adequate in expressing a wide range of deterministic parallel algorithms clearly and concisely [Hudak 1986c; Hudak and Smith 1986]. It remains to be seen, however, whether the pragmatic concerns that motivate these kinds of language extensions persist, and if they do, whether or not compilers can become smart enough to perform the optimizations automatically. Indeed these same questions can be asked about other language extensions, such as the memoization techniques discussed in the next section.

3.6 Caching and Memoization

Consider this simple definition of the Fibonacci function:

```
fib 0 = 1
'   1 = 1
'   n = fib (n−1) + fib (n−2)
```

Although simple, it is hopelessly inefficient. We could rewrite it in one of the classic ways, but then the simplicity and elegance of the original definition is lost. Keller and Sleep [1986] suggest an elegant alternative: Provide syntax for expressing the *caching* or *memoization* of selected functions. For example, the syntax might take the form of a declaration that precedes the function definition, as in

```
memo fib using cache
fib 0 = 1
'   1 = 1
'   n = fib (n−1) + fib (n−2)
```

which would be syntactic sugar for

```
fib = cache fib1
  where fib1 0 = 1
        '    1 = 1
        '    n = fib (n−1) + fib (n−2)
```

The point is that cache is a user-defined function that specifies a strategy for caching values of fib. For example, to cache values in an array, we might define cache by

```
cache fn = \n → (array (0,max)
                  [(i,fn i) | i←[0..max]]) ! n
```

where we assume max is the largest argument to which fib will be applied. Expanding out the definitions and syntax yields

```
fib n = (array (0,max)
          [(i,fib1 i) | i←[0..max]]) ! n
  where fib1 0 = 1
        '    1 = 1
        '    n = fib (n−1) + fib (n−2)
```

which is exactly the desired result.[20] As a methodology this is very nice, since libraries of useful caching functionals may be produced and reused to cache many different functions. There are limitations to the approach, as well as extensions, all of which are described in Keller and Sleep [1986].

One of the limitations of this approach is that in general it can only be used with strict functions, and even then the expense of performing equality checks on, for example, list arguments can be expensive. As a solution to this problem. Hughes introduced the notion of *lazy memo-functions*, in which the caching strategy uses an identity test (EQ in Lisp terminology) instead of an equality test [Hughes 1985b]. Such a strategy can no longer be considered as syntactic sugar, since an identity predicate is not something normally provided as a primitive in functional languages because it is implementation dependent. Nevertheless, if built into a language lazy memo-functions provide a very efficient (constant time) caching mechanism and allow very elegant solutions to a class of problems not solved by Keller and Sleep's strategy: those involving infinite data structures. For example, consider this definition of the infinite list of ones:

```
ones = 1 : ones
```

Any reasonable implementation will represent this as a cyclic list, thus consuming constant space. Another common idiom is the use of higher-order functions such as

[20] This is essentially the same solution as the one given in Section 3.3.

map:

```
twos = map (\x→2*x) ones
```

But now note that only the cleverest of implementations will represent this as a cyclic list, since map normally generates a new list cell on every recursive call. By lazy memoizing map, however, the cyclic list will be recovered. To see how, note that the first recursive call to map will be "map (\x→ 2*x) (tail ones)"—but (tail ones) is identical to ones (remember that ones is cyclic), and thus map is called with arguments identical to the first call. Thus the old value is returned, and the cycle is created. Many more practical examples are discussed in Hughes [1985b].

The interesting thing about memoization in general is that it begins to touch on some of the limitations of functional languages—in particular, the inability to side effect global objects such as caches—and solutions such as lazy memo-functions represent useful compromises. It remains to be seen whether more general solutions can be found that eliminate the need for these special-purpose features.

3.7 Nondeterminism

Most programmers (including the very idealistic among us) admit the need for nondeterminism, despite the semantic difficulties it introduces. It seems to be an essential ingredient of real-time systems, such as operating systems and device controllers. Nondeterminism in imperative languages is typically manifested by running in parallel several processes that are side effecting some global state—the nondeterminism is thus implicit in the semantics of the language. In functional languages, nondeterminism is manifested through the use of primitive operators such as amb or merge—the nondeterminism is thus made explicit. Several papers have been published on the use of such primitives in functional programming, and it appears quite reasonable to program conventional nondeterministic applications using them [Henderson 1982; Stoye 1985]. The problem is, once introduced, nondeterminism completely destroys referential transparency, as we shall see.

By way of introduction, McCarthy [1963] defined a binary nondeterministic operator called *amb* having the following behavior:

$$amb(e_1, \perp) = e_1$$
$$amb(\perp, e_2) = e_2$$
$$amb(e_1, e_2) = \text{either } e_1 \text{ or } e_2, \text{ chosen nondeterministically}$$

The operational reading of $amb(e_1, e_2)$ is that e_1 and e_2 are evaluated in parallel, and the one that completes first is returned as the value of the expression.

To see how referential transparency is lost, consider this simple example:

```
(amb 1 2) + (amb 1 2)
```

Is the answer 2 or 4? Or is it perhaps 3? The possibility of the answer 3 indicates that referential transparency is lost—there does not appear to be any simple syntactic mechanism for ensuring that we could not replace equals for equals in a misleading way. Shortly, we will discuss possible solutions to this problem, but for now let us look at an example using this style of nondeterminism.

Using *amb*, we can easily define things such as merge that nondeterministically merge two lists, or streams:[21]

```
merge as bs = amb
    (if (as == [ ]) then bs else
                (head as : merge (tail as) bs))
    (if (bs == [ ]) then as else
                (head bs : merge as (tail bs)))
```

which then can be used, for example, in combining streams of characters from different computer terminals:

```
process (merge term1 term2)
```

[21] Note that this version of merge:

```
merge [ ] bs = bs
merge as [ ] = as
merge (a:as) (b:bs)
  = amb (a : merge as (b:bs)) (b : merge (a:as)
  bs)
```

is not correct, since merge ⊥ *bs* evaluates to ⊥, whereas we would like it to be *bs* ++ ⊥, which in fact the definition in the text yields.

Using this as a basis, Henderson [1982] show how many operating system problems can be solved in a pseudofunctional language. Hudak [1986a] uses nondeterminism of a slightly different kind to emulate the parallel updating of arrays.

Although satisfying in the sense of being able to solve real-world kinds of nondeterminism, these solutions are dissatisfying in the way they destroy referential transparency. One might argue that the situation is at least somewhat better than the conventional imperative one in that the nondeterminism is at least made explicit, and thus one could induce extra caution when reasoning about those sections of a program exhibiting nondeterministic behavior. The only problem with this is that determining which sections of a program are nondeterministic may be difficult—it is not a lexical property, but rather a dynamic one, since any function may call a nondeterministic subfunction.

At least two solutions have been proposed to this problem. One, proposed by Burton [1988], is to provide a tree-shaped oracle as an argument to a program from which nondeterministic choices may be selected. By passing the tree and its subtrees around explicitly, referential transparency can be preserved. The problem with this approach is that carrying the oracle around explicitly is cumbersome at best. On the other hand, functional programmers already carry around a greater amount of state to avoid problems with side effects, so perhaps the extra burden is not too great.

Another (at least partial) solution was proposed by Stoye [1985] in which all of the nondeterminism in a program is forced to be in one place. Indeed to some extent this was the solution adopted in Haskell, although for somewhat different reasons. The problem with this approach is that the nondeterminism is not eliminated completely but rather centralized. It allows reasoning equationally within the isolated pieces but not within the collection of pieces as a whole. Nevertheless, the isolation (at least in Haskell) is achieved syntactically, and thus it is easy to determine when equational reasoning is valid. A general discussion of these issues is found in Hudak and Sundaresh [1988].

An interesting variation on these two ideas is to combine them—centralize the nondeterminism and then use an oracle to define it in a referentially transparent way. Thus the disadvantages of both approaches would seem to disappear.

In any case, I should point out that none of these solutions makes reasoning about nondeterminism any easier, they just make reasoning about programs easier.

3.8 Extensions to Polymorphic-Type Inference

The Hindley–Milner type system has certain limitations; an example of this was given in Section 1.6.1. Some success has been achieved in extending the type system to include other kinds of data objects, but surprisingly little success has been achieved at removing the fundamental limitations while still retaining the tractability of the type inference problem. It is a somewhat fragile system but is fortunately expressive enough to base practical languages on it. Nevertheless, research continues in this area.

Independently of type inference, considerable research is underway on the expressiveness of type systems in general. The most obvious thing to do is allow types to be first class, thus allowing abstraction over them in the obvious way. Through generalizations of the type system it is possible to model such things as parameterized modules, inheritance, and subtyping. This area has indeed taken on a character of its own; a good summary of current work may be found in Cardelli and Wegner [1985] and Reynolds [1985].

3.9 Combining Other Programming Language Paradigms

A time-honored tradition in programming language design is to come up with hybrid designs that combine the best features of several different paradigms, and functional programming language research has not escaped that tradition. I will discuss two such hybrids here, although others exist.

The first hybrid is combining logic programming with functional programming. The "logical variable permits" two-way matching (via unification) in logic

programming languages, as opposed to the one-way matching (via pattern matching) in functional languages, and thus seems like a desirable feature to have in a language. Indeed its declarative nature fits well with the ideals of functional programming. The integration is, however not easy—many proposals have been made yet none are completely satisfactory, especially in the context of higher order functions and lazy evaluation. See Degroot and Lindstrom [1985] for a good summary of results.

The second area reflects an attempt to combine the state-oriented behavior of imperative languages in a semantically clean way. The same problems arise here as they do with nondeterminism—simply adding the assignment statement means that equational reasoning must always be qualified, since it is difficult to determine whether or not a deeply nested call is made to a function that induces a side effect. There are, however, some possible solutions to this, most notably work by Gifford and Lucassen [Gifford and Lucassen 1986; Lucassen and Gifford 1988] in which effects are captured in the *type system.* In Gifford's system it is possible to determine from a procedure's type whether or not it is side-effect free. It is not currently known, however, whether such a system can be made into a type inference system in which type declarations are not required. This is an active area of current research.

4. Dispelling Myths About Functional Programming

To gain further insight into the nature of functional languages, it is helpful to discuss, with the hope of dispelling, certain myths that have arisen over the years.

Myth 1, that functional programming is the antithesis of conventional imperative programming, is largely responsible for alienating imperative programmers from functional languages. But, in fact, there is much in common between the two styles of programming, which I make evident by two simple arguments.

Consider first that one of the key evolutionary characteristics of high-level imperative programming languages has been the use of expressions to denote a result rather than a sequence of statements to put together a result imperatively and in piecemeal. Expressions were an important feature of FORTRAN, had more of a mathematical flavor, and freed the programmer of low-level operational detail (this burden was of course transferred to the compiler). From FORTRAN expressions, to functions in Pascal, to expression oriented programming style in Scheme–these advances are all on the same evolutionary path. Functional programming can be seen as carrying this evolution to its logical conclusion—*everything* is an expression.

The second argument is based on an analogy between functional (i.e., side-effect-free) programming and structured (i.e., goto-less) programming. The fact is, it is hard to imagine doing without either goto's or assignment statements, until one is shown what to use in their place. In the case of goto, one uses instead structured commands, in the case of assignment statements, one uses instead lexical binding and recursion.

As an example, this simple program fragment with goto's

```
        x := init;
        i := 0;
loop:   x := f(x, i);
        i := i+1;
        if i<10 goto loop;
```

can be rewritten in a structured style as

```
x := init;
i := 0;
while i<10
  being x := f(x, i);
        i := i+1
  end;
```

In capturing this disciplined use of goto, arbitrary jumps into or out of the body of the block now cannot be made. Although this can be viewed as a constraint, most people feel that the resulting disciplined style of programming is clearer, easier to maintain, and so on.

More discipline is evident here than just the judicious use of goto. Note in the original program fragment that x and i are assigned to exactly once in each iteration of the loop; the variable i, in fact, is only being used to control the loop termination

criteria, and the final value of x is intended as the value computed by the loop. This disciplined use of assignment can be captured by the following assignment-free Haskell program:

```
loop init 0
where loop x i = if i<10
                 then (loop (f x i) (i+1)
                 else x
```

Functions (and procedures) can in fact be thought of as a disciplined use of goto and assignment—the transfer of control to the body of the function and the subsequent return capture a disciplined use of goto, and the formal-to-actual parameter binding captures a disciplined use of assignment.

By inventing a bit of syntactic sugar to capture the essence of tail recursion, the above program could be rewritten as

```
let x = init
    i = 0
in while i<10
   begin next x = f(x, i)
         next i = i+1
   end
   result x
```

[this syntactic sugar is not found in Haskell, although some other functional (especially dataflow) languages have similar features, including Id, Val, and Lucid] where the form "next $x = \ldots$" is a construct (next is a keyword) used to express what the value of x will be on the next iteration of the loop. Note the similarity of this program to the structured one given earlier. In order to enforce a disciplined use of assignment properly, we can constrain the syntax so that only one next statement is allowed for each identifier (stated another way, this constraint means that it is a trivial matter to convert such programs into the tail recursive form shown earlier). If we think of "next x" and "next i" as new identifiers (just as the formal parameters of loop can be thought of as new identifiers for every call to loop), then referential transparency is preserved. Functional programming advocates argue that this results in a better style of programming, in much the same way that structured programming advocates argue for their cause.

Thus the analogy between goto-less programming and assignment-free programming runs deep. When Dijkstra first introduced structured programming, much of the programming community was aghast—how could one do without goto? But as people programmed in the new style, it was realized that what was being imposed was a discipline for good programming not a police state to inhibit expressiveness. Exactly the same can be said of side-effect-free programming, and its advocates hope that as people become more comfortable programming in the functional style, they will appreciate the good sides of the discipline thus imposed.

When viewed in this way functional languages can be seen as a logical step in the evolution of imperative languages—thus, of course, rendering them nonimperative. On the other hand, it is exactly this purity to which some programmers object, and one could argue that just as a tasteful use of goto here or there is acceptable, so is a tasteful use of a side effect. Such small impurities certainly shouldn't invalidate the functional programming style and thus may be acceptable.

Myth 2 is that functional programming languages are toys. The first step toward dispelling this myth is to cite examples of efficient implementations of functional languages, of which there now exist several. The Alfl compiler at Yale, for example, generates code that is competitive with that generated by conventional language compilers [Young 1988]. Other notable compiler efforts include the LML compiler at Chalmers University [Augustsson 1984], the Ponder compiler at Cambridge University [Fairbairn 1985], and the ML compilers developed at Bell Labs and Princeton [Appel and MacQueen 1987].

On the other hand, there are still inherent inefficiencies that cannot be ignored. Higher-order functions and lazy evaluation certainly increase expressiveness, but in the general case the overhead of, for example, the dynamic storage management necessary to support them cannot be eliminated.

The second step toward dispelling this myth amounts to pointing to real applications development in functional languages

including real-world situations involving nondeterminism, databases, parallelism, and so on. Atlhough examples of this sort are not plentiful (primarily because of the youth of the field) and are hard to cite (since papers are not usually written about applications), they do exist. For example,

(1) The dataflow groups at MIT and the functional programming groups at Yale have written numerous functional programs for scientific computation. So have two national labs: Los Alamos and Lawrence Livermore.
(2) MCC has written a reasonably large expert system (EMYCIN) in SASL.
(3) At least one company is currently marketing a commercial product (a CAD package) that uses a lazy functional language.
(4) A group at IBM uses a lazy functional language for graphics and animation.
(5) The LML (lazy ML) compiler at Chalmers was written almost entirely in LML, and the new Haskell compilers at both Glasgow and Yale are being written in Haskell.
(6) GEC Hirst Research Lab is using a program for designing VLSI circuits that was written by some researchers at Oxford using a lazy functional language.

There are other examples. In particular, there are many Scheme and Lisp programs that are predominantly side effect free and could properly be thought of as functional programs.

Myth 3, that functional languages cannot deal with state, is often expressed as a question: How can someone program in a language that does not have a notion of state? The answer, of course, is that we cannot, and in fact functional languages deal with state very nicely, although the state is expressed explicitly rather than implicitly. So the issue is more a matter of how one expresses state and manipulations of it.

State in a functional program is usually carried around explicitly in one of two ways: (1) in the values of bound variables of functions, in which case it is updated by making a recursive call to the function with new values as arguments, or (2) in a data structure that is updated nondestructively so that, at least conceptually, the old value of the data structure remains intact and can be accessed later. Although declarative and referentially transparent, this treatment of state can present problems to an implementation, but it is certainly not a problem in expressiveness. Furthermore, the implementation problems are not as bad as they first seem, and recent work has gone a long way toward solving them. For example, a compiler can convert tail recursions into loops and "single-threaded" data structures into mutable ones.

It turns out that, with respect to expressiveness, one can use higher-order functions not only to manipulate state but also to make it appear implicit. To see how, consider the imperative program given earlier:

```
      x := init;
      i := 0;
loop: x := f(x, i);
      i := i + 1;
      if (i < 10) goto loop;
```

We can model the implicit state in this program as a pair (xval, ival) and define several functions that update this state in a way that mimics the assignment statements:

```
x (xval, ival) xval' = (xval', ival)
i (xval, ival) ival' = (xval, ival')
x' (x, i) = x
i' (x, i) = i
const v s = v
```

We will take advantage of the fact that these functions are defined in curried form.

Note how x and i are used to update the state, and x' and i' are used to access the state. For example, the following function, when applied to the state, will increment i:

```
\s → i s ((i' s) + 1)
```

For expository purposes we would like to make the state as implicit as possible, and thus we express the result as a composition of higher-order functions. To facilitate this and to make the result look as much like the original program as possible, we define the following higher-order infix operators

and functions[22]:

```
f := g  = \s → f s (g s)
f ; g   = \s → g (f s)
goto f  = f
f +' g  = \s → f s + g s
f <' g  = \s → f s < g s
if' p c = \s → (if (p s) then (c s) else s)
```

[I am cheating slightly here in that ; is a reserved operator in Haskell and thus cannot really be redefined in this way.]

Given these definitions, we can now write the following functional (albeit contrived) version of the imperative program given earlier:

```
    x := const init;
    i := const 0;
loop where
loop = x := f;
       i := i' +' const 1;
       if' (i' <' const 10) (goto loop)
```

This result is rather disquieting—it looks very much like the original imperative program. Of course, we worked hard to make it that way, which in the general case is much harder to do and it is certainly not the recommended way to do functional programming. Nevertheless it exemplifies the power and flexibility of higher-order functions—note how they are used here both to manipulate the state and to implement the goto (where in particular the definition of loop is recursive, since the goto implements a loop).

5. Conclusions

This paper presented functional programming in its many shapes and forms. Although it only touched on the surface of many issues, it is hoped that enough of a foundation has been given that researchers can explore particularly interesting topics in more depth and programmers can learn to use functional languages in a variety of applications.

I said little about how to implement functional languages, primarily because doing that subject justice would probably double the size of this paper. The interested reader can refer to by far the best single reference to sequential implementations, Peyton Jones' [1987], as well as other techniques that have recently appeared viable [Bloss et al. 1988; Burn et al. 1988; and Fairbairn and Wray 1987]. Parallel implementations have taken a variety of forms. On commercial machines the state of the art on parallel graph reduction implementations may be found in Goldberg and Hudak [1988] and Goldberg [1988a, 1988b]. The latest on special-purpose parallel graph reducers can be found in Peyton Jones et al. [1987] and Watson and Watson [1987]. For a different kind of implementation see Hudak and Mohr [1988]. References to dataflow machines were given in Section 1.8.

[22] It is interesting to note that :=, const, and goto correspond precisely to the combinators *S*, *K*, and *I*, and ; is almost the combinator *B*, but is actually *CB*.

ACKNOWLEDGMENTS

Thanks to the Lisp and Functional Programming Research Group at Yale, which has inspired much of my work and served as my chief sounding board. A special thanks is also extended to the Haskell Committee, through which I learned more than I care to admit. In addition, the following people provided valuable comments on early drafts of the paper: Kei Davis, Alex Ferguson, John Launchbury, and Phil Wadler (all from the University of Glasgow); Juan Guzman, Siau-Cheng Khoo, Amir Kishon, Raman Sundaresh, and Pradeep Varma (all from Yale); David Wise (Indiana University); and three anonymous referees.

I also wish to thank my funding agencies: The Department of Energy under grant FG02-86ER25012, the Defense Advanced Research Projects Agency under grant N00014-88-K-0573, and the National Science Foundation under grant DCR-8451415. Without their generous support none of this work would have been possible.

In writing this survey paper I have tried to acknowledge accurately the significant technical contributions in each of the areas of functional programming that I have covered. Unfortunately, that is a very difficult task, and I apologize in advance for any errors or omissions.

REFERENCES

AASA, A., HOLMSTROM, S., AND NILSSON, C. 1987. An efficiency comparison of some representations of purely functional arrays. Tech. Rep. 33. Programming Methodology Group, Chalmers University of Technology.

ABELSON, H., SUSSMAN, G. J., AND SUSSMAN, J. 1985. *Structure and Interpretation of Computer*

Programs. The MIT Press, Cambridge, Mass., and McGraw-Hill New York.

ACKERMAN, W. B., AND DENNIS, J. B. 1979. VAL—A value-oriented algorithmic language preliminary reference manual. Laboratory for Computer Science MIT/LCS/TR-218, MIT.

ANDERSON, S., AND HUDAK, P. 1989. Efficient compilation of Haskell array comprehensions. Tech. Rep. YALEU/DCS/RR693. Yale University, Department of Computer Science.

APPEL, A. W., AND MACQUEEN, D. B. 1987. A standard ML compiler. In *Proceedings of the 1987 Functional Programming Languages and Computer Architecture Conference* (Sept. 1987). Springer-Verlag LNCS 274, IFIP pp. 301–324.

ARVIND AND GOSTELOW, K. P. 1977. A computer capable of exchanging processors for time. In *Proceedings IFIP Congress*, pp. 849–853.

ARVIND AND GOSTELOW, K. P. 1982. The U-interpreter. *Computer 15*, 2, 42–50.

ARVIND AND KATHAIL, V. 1981. A multiple processor data flow machine that supports generalized procedures. In *Proceedings of the 8th Annual Symposium on Computer Architecture*. Vol. 9, No. 3. ACM SIGARCH, pp. 291–302.

ASHCROFT, E. A., AND WADGE, W. W. 1976a. Lucid—A formal system for writing and proving programs. *SIAM J. Comput.* 5, 3, 336–354.

ASHCROFT, E. A., AND WADGE, W. W. 1976b. Lucid, a nonprocedural language with iteration. *Commun. ACM 20*, 519–526.

AUGUSTSSON, L. 1984. A compiler for Lazy ML. In *Proceedings 1984 ACM Conference on LISP and Functional Programming* (August). ACM, pp. 218–227.

AUGUSTSSON, L. 1985. Compiling pattern-matching. In *Functional Programming Languages and Computer Architecture*. Springer-Verlag LNCS 201, pp. 368–381.

BACKUS, J. 1978. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8, 613–641.

BACKUS, J., WILLIAMS, J. H., AND WIMMERS, E. L. 1986. FL language manual (preliminary version). Tech. Rep. RJ 5339 (54809). Computer Science, IBM Almaden Research Center, Almaden, CA.

BARENDREGT, H. P. 1984. *The Lambda Calculus, Its Syntax and Semantics*. Revised ed. North-Holland, Amsterdam.

BERRY, G. 1978. Séquentialité de l'évaluation formelle des λ-expressions. In *Proceedings 3-e Colloque International sur la Programmation*.

BIRD, R., AND WADLER, P. 1988. *Introduction to Functional Programming*. Prentice Hall, Englewood Cliffs, N.J.

BLOSS, A. 1988. Path analysis: Using order-of-evaluation information to optimize lazy functional languages. Ph.D. Dept. Computer Science, dissertation, Yale Univ.

BLOSS, A., AND HUDAK, P. 1987. Path semantics. In *Proceedings of Third Workshop on the Mathematical Foundations of Programming Language Semantics*. Springer-Verlag LNCS (Tulane Univ., April 1987), 298, 476–489.

BLOSS, A., HUDAK, P., AND YOUNG, J. 1988. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation: An International Journal 1*, 147–164.

BOEHM, H.-J. 1985. Partial polymorphic type inference is undecidable. In *Proceedings of 26th Sympsoium on Foundations of Computer Science*. IEEE pp. 339–345.

BOUTEL, B. E. 1988. Tui language manual. Tech. Rep. CSD-8-021. Victoria University of Wellington, Department of Computer Science.

BURGE, W. H. 1975. *Recursive Programming Techniques*. Addison-Wesley, Reading, Mass.

BURN, G. L., PEYTON JONES, S. L., AND ROBSON, J. D. 1988. The spineless G-machine. In *Proceedings 1988 ACM Conference on Lisp and Functional Programming* (Salt Lake City, Utah). ACM SIGPLAN/SIGACT/SIGART, 244–258.

BURSTALL, R. M., MACQUEEN, D. B., AND SANNELLA, D. T. 1980. HOPE: An experimental applicative language. In *The 1980 LISP Conference*. Stanford University, Santa Clara Univ. The USP Co., pp. 136–143.

BURTON, F. W. 1984. Annotations to control parallelism and reduction order in the distributed evaluation of functional programs. *ACM Trans. Program. Lang. Syst. 6*, 2, 159–174.

BURTON, F. W. 1988. Nondeterminism with referential transparency in functional programming languages. *Comput. J. 31*, 3, 243–247.

CARDELLI, L., AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4, 471–522.

CARTWRIGHT, R. 1976. A practical formal semantic definition and verification system for typed Lisp. Tech. Rep. AIM-296. Stanford Artificial Intelligence Laboratory.

CHEN, M. C. 1986. Transformations of parallel programs in crystal. In *Information Processing '86*, Elsevier North-Holland, New York, pp. 455–462.

CHURCH, A. 1932–1933. A set of postulates for the foundation of logic. *Ann. Math. 2*, 33–34, 346–366, 839–864.

CHURCH, A. 1941. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, N.J.

CHURCH, A., AND ROSSER, J. B. 1936. Some properties of conversion. *Trans. Am. Math. Soc. 39*, 472–482.

CURRY, H. B., AND FEYS, R. 1958. *Combinatory Logic*. Vol. 1. North-Holland, The Netherlands.

DAMAS, L., AND MILNER, R. 1982. Principle type schemes for functional languages. In *9th ACM Symposium on Principles of Programming Languages*. ACM.

DARLINGTON, J., AND WHILE, L. 1987. Controlling the behavior of functional language systems. In

Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference. Springer-Verlag LNCS 274, pp. 278–300.

DAVIS, A. L. 1978. The architecture and system method of DDM-1: A recursively structured data driven machine. In *Proceedings 5th Annual Symposium on Computer Architecture.* IEEE, ACM.

DEGROOT, D., AND LINDSTROM, G. 1985. *Functional and Logic Programming.* Prentice-Hall, Englewood Cliffs, N.J.

DELOSME, J.-M., AND IPSEN, I. C. F. 1985. An illustration of a methodology for the construction of efficient systolic architectures in VLSI. In *Proceedings 2nd International Symposium on VLSI Technology, Systems, and Applications,* ITRI, NSC pp. 268–273.

DENNIS, J. B., AND MISUNAS, D. P. 1974. A preliminary architecture for a basic dataflow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture.* ACM, IEEE, pp. 126–132.

FAIRBAIRN, J. 1985. Design and implementation of a simple typed language based on the lambda calculus. Ph.D. dissertation, Univ. of Cambridge. Available as Computer Laboratory TR No. 75.

FAIRBAIRN, J., AND WRAY, S. 1987. Tim: A simple, lazy abstract machine to execute supercombinators. In *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference.* Springer Verlag LNCS 274, pp. 34–45.

FIELD, A. J., AND HARRISON, P. G. 1988. *Functional Programming.* Addison-Wesley, Workingham, England.

FORTUNE, S., LEIVANT, D., AND O'DONNELL, M. 1985. The expressiveness of simple and second-order type structures. *J. ACM 30,* 1, 151–185.

FRIEDMAN, D. P., AND WISE, D. S. 1976. Cons should not evaluate its arguments. In *Automata, Languages and Programming,* Edinburgh University Press, pp. 257–284.

GELERNTER, H., HANSEN, J. R., AND GERBERICH, C. L. 1960. A FORTRAN-compiled list processing language. *J. ACM 7,* 2, 87–101.

GIFFORD, D. K., AND LUCASSEN, J. M. 1986. Integrating functional and imperative programming. In *Proceedings 1986 ACM Conference on Lisp and Functional Programming.* ACM SIGPLAN/SIGACT/SIGART, pp. 28–38.

GIRARD, J.-Y. 1972. Interpretation Fonctionelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur. Ph.D. dissertation, Univ. of Paris.

GOLDBERG, B. 1988a. Buckwheat: Graph reduction on a shared memory multiprocessor. In *Proceedings 1988 ACM Conference on Lisp and Functional Programming* (Salt Lake City, Utah, August 1988) ACM SIGPLAN/SIGACT/SIGART.

GOLDBERG, B. 1988b. Multiprocessor execution of functional programs. Ph.D. dissertation, Dept. of Computer Science, Yale Univ. Available as Tech. Rep. YALEU/DCS/RR-618.

GOLDBERG, B., AND HUDAK, P. 1988. Implementing functional programs on a hypercube multiprocessor. In *Proceedings of 3rd Conference on Hypercube Concurrent Computers and Applications.* ACM.

GORDON, M. J., MILNER, R., AND WADSWORTH, C. P. 1979. *Edinburgh LCF.* Springer-Verlag LNCS 78, Berlin.

GORDON, M., MILNER, R., MORRIS, L., NEWEY, M., AND WADSWORTH, C. 1978. A metalanguage for interactive proof in LCF. In *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages.* ACM, pp. 119–130.

GUTTAG, J., HORNING, J., AND WILLIAMS, J. 1981. FP with data abstraction and strong typing. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture.* ACM, pp. 11–24.

HANCOCK, P. 1987. Polymorphic type-checking. In *The Implementation of Functional Programming Languages,* S. L. Peyton Jones, Ed. Prentice-Hall International, Englewood Cliffs, N.J., Chapters 8 and 9.

HENDERSON, P. 1980. *Functional Programming: Application and Implementation.* Prentice-Hall, Englewood Cliffs, N.J.

HENDERSON, P. 1982. Purely functional operating systems. In *Functional Programming and Its Applications: An Advance Course.* Cambridge University Press, pp. 177–192.

HENDERSON, P. AND MORRIS, L. 1976. A lazy evaluator. In *3rd ACM Symposium on Principles of Programming Languages.* ACM, pp. 95–103.

HINDLEY, R. 1969. The principle type scheme of an object in combinatory logic. *Trans. Amer. Math. Soc. 146,* 29–60.

HOLMSTROM, S. 1983. How to handle large data structures in functional languages. In *Proceedings of SERC/Chalmers Workshop on Declarative Programming Languages.* SERC.

HUDAK, P. 1984. *ALFL Reference Manual and Programmer's Guide.* 2nd ed. Res. Rep. YALEU/DCS/RR-322. Yale University.

HUDAK, P. 1986a. Arrays, non-determinism, side-effects, and parallelism: a functional perspective. In *Proceedings of the Santa Fe Graph Reduction Workshop* (Los Alamos/MCC). Springer-Verlag LNCS 279, pp. 312–327.

HUDAK, P. 1986b. Denotational semantics of a para-functional programming language. *Int. J. Parallel Program. 15,* 2, 103–125.

HUDAK, P. 1986. Para-functional programming. *Computer 19,* 8, 60–71.

HUDAK, P., AND ANDERSON, S. 1987. Pomset interpretations of parallel functional programs. In *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference.* Springer Verlag LNCS 274, pp. 234–256.

HUDAK, P., AND ANDERSON, S. 1988. Haskell solutions to the language session problems at the 1988 Salishan high-speed computing conference.

Tech. Rep. YALEU/DCS/RR-627. Department of Computer Science, Yale University.

HUDAK, P., AND MOHR, E. 1989. Graphinators and the duality of SIMD and MIMD. In *Proceedings 1988 ACM Conference on Lisp and Functional Programming* (Salt Lake City, Utah, August). ACM SIGPLAN/SIGACT/SIGART.

HUDAK, P., AND SUNDARESH, R. 1988. On the expressiveness of purely functional I/O systems. Tech. Rep. YALEU/DCS/RR-665. Department of Computer Science. Yale University.

HUDAK, P., AND SMITH, L. 1986. Para-functional programming: A paradigm for programming multiprocessor systems. In *12th ACM Symposium on Principles of Programming Languages*. ACM, pp. 243–254.

HUDAK, P., AND WADLER, P. Eds. 1988. Report on the Functional Programming Language Haskell. Tech. Rep. YALEU/DCS/RR656. Department of Computer Science, Yale University.

HUGHES, J. 1984. Why functional programming matters. Tech. Rep. 16. Programming Methodology Group, Chalmers University of Technology.

HUGHES, J. 1985a. An efficient implementation of purely functional arrays. Tech. Rep. Programming Methodology Group, Chalmers University of Technology.

HUGHES, J. 1985b. Lazy memo-functions. In *Functional Programming Languages and Computer Architecture*. Springer-Verlag LNCS 201, pp. 129–146.

IVERSON, K. 1962. *A Programming Language*. Wiley, New York.

JOHNSON, S. D. 1988. Daisy Programming Manual. Tech. Rep. Indiana University Computer Science Department.

KAES, S. 1988. Parametric polymorphism. In *Proceedings of the 2nd Eupropean Symposium on Programming*. Springer-Verlag LNCS 300.

KELLER, R. M. 1982. FEL programmer's guide. AMPS TR 7. University of Utah.

KELLER, R. M., AND LINDSTROM, G. 1985. Approaching distributed database implementations through functional programming concepts. In *International Conference on Distributed Systems*. IEEE.

KELLER, R. M., AND SLEEP, R. 1986. Applicative caching. *ACM Trans. Program. Lang. Syst. 8*, 1, 88–108.

KELLER, R. M., JAYARAMAN, B., ROSE, D., AND LINDSTROM, G. 1980. FGL programmer's guide. AMPS Tech. Memo 1. Department of Computer Science, University of Utah.

KLEENE, S. C. 1936. λ-definability and recursiveness. *Duke Math. J. 2*, 340–353.

KLEENE, S. C., AND ROSSER, J. B. 1935. The inconsistency of certain forms of logic. *Ann. Math. 2*, 36, 630–636.

KROEZE, H. J. 1986–1987. The TWENTEL system (version 1). Tech. Rep. Department of Computer Science, University of Twente, The Netherlands.

LANDIN, P. J. 1964. The mechanical evaluation of expressions. *Comput. J. 6*, 4, 308–320.

LANDIN, P. J. 1965. A correspondence between ALGOL 60 and Church's lambda notation. *Commun. ACM 8*, 89–101, 158–165.

LANDIN, P. J. 1966. The next 700 programming languages. *Commun. ACM 9*, 3, 157–166.

LUCASSEN, J. M., AND GIFFORD, D. K. 1988. Polymorphic effect systems. In *Proceedings of 15th ACM Symposium on Principles of Programming Languages*. ACM, pp. 47–57.

MARKOV, A. A. 1951. Teoriya algorifmov (Theory of algorithms). *Trudy Mat. Inst. Steklov 38*, 176–189.

MCCARTHY, J. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM 3*, 4, 184–195.

MCCARTHY, J. 1963. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*. North-Holland, The Netherlands, pp. 33–70.

MCCARTHY, J. 1978. History of Lisp. In *Preprints of Proceedings of ACM SIGPLAN History of Programming Languages Conference*. SIGPLAN Notices, Vol. 13, pp. 217–223.

MCGRAW, J. R. 1982. The VAL language: Description and analysis. *TOPLAS, 4*, 1, 44–82.

MCGRAW, J., ALLAN, S., GLAUERT, J., AND DOBES, I. 1983. SISAL: Streams and Iteration in a Single-Assignment Language, Language Reference Manual. Tech. Rep. M-146. Lawrence Livermore National Laboratory.

MILNE, R. E., AND STRACHEY, C. 1976. *A Theory of Programming Language Semantics*. Chapman and Hall, London, and John Wiley, New York.

MILNER, R. A. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 3, 348–375.

MILNER, R. 1984. A proposal for Standard ML. In *Proceedings 1984 ACM Conference on LISP and Functional Programming*. ACM, pp. 184–197.

MULLIN, L. R. 1988. A mathematics of arrays. Ph.D dissertation, Computer and Information Science and CASE Center, Syracuse University.

NIKHIL, R. S., PINGALI, K., AND ARVIND. 1986. Id nouveau. Computation Structures Group Memo 265. Laboratory for Computer Science, Massachusetts Institute of Technology.

PEYTON JONES, S. L. 1987. *The Implementation of Functional Programming Languages*. Prentice-Hall International, Englewood Cliffs, N.J.

PEYTON JONES, S. L., CLACK, C., SALKILD, J., AND HARDIE, M. GRIP—A high-performance architecture for parallel graph reduction. In *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference*. Springer-Verlag LNCS 274, pp. 98–112.

PFENNING, F. 1988. Partial polymorphic type inference and higher-order unification. In *Proceedings 1988 ACM Conference on Lisp and Functional*

Programming (Salt Lake City, Utah). ACM SIGPLAN/SIGACT/SIGART, pp. 153–163.

POST, E. L. Formal reductions of the general combinatorial decision problem. *Am. J. Math. 65*, 197–215.

REES, J., AND CLINGER, W. Eds. 1986. The revised report on the algorithmic language Scheme. *SIGPLAN Notices 21*, 12, 37–79.

REYNOLDS, J. C. 1974. Towards a theory of type structure. In *Proceedings of Colloque sur la Programmation*. Springer-Verlag LNCS 19, pp. 408–425.

REYNOLDS, J. C. 1985. Three approaches to type structure. In *Mathematical Foundations of Software Development*, Springer-Verlag LNCS 185, pp. 97–138.

ROSSER, J. B. 1982. Highlights of the history of the lambda-calculus. In *Proceedings 1982 ACM Conference on LISP and Functional Programming*. ACM, pp. 216–225.

SCHMIDT, D. A. 1985. Detecting global variables in denotational specifications. *ACM Trans. Program. Lang. Syst. 7*, 2, 299–310.

SCHÖNFINKEL, M. 1924. Uber die bausteine der mathematischen logik. *Mathematische Annalen 92*, 305.

SCOTT, D. S. 1970. Outline of a mathematical theory of computation. Programming Research Group PRG-2, Oxford University.

SHAPIRO, E. 1989. *Systolic Programming: A Paradigm of Parallel Processing*. Department of Applied Mathematics Tech. Rep. CS84-21, The Weizmann Institute of Science.

SRIDHARAN, N. S. 1985. Semi-applicative programming: An example. Tech. Rep. BBN Laboratories.

STEELE, JR., L. G., AND HILLIS, D. W. 1986. Connection machine lisp: Fine-grained parallel symbolic processing. In *Proceedings 1986 ACM Conference on Lisp and Functional Programming* (Cambridge, Mass.). ACM SIGPLAN/SIGACT/SIGART, pp. 279–297.

STOY, J. E. 1977. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Mass.

STOYE, W. 1985. A New Scheme for Writing Functional Operating Systems. Tech. Rep. 56. Computer Laboratory, University of Cambridge.

THAKKAR, S. S. Ed. 1987. *Selected Reprints on Dataflow and Reduction Architectures*. The Computer Society Press, Washington, DC.

TOFTE, M. 1988. Operational semantics and polymorphic type inference. Ph.D. dissertation, Dept. Computer Science, Univ. of Edinburgh (CST-52-88).

TRAKHTENBROT, B. A. 1988. Comparing the Church and Turing approaches: Two prophetic messages. Tech. Rep. 98/88. Eskenasy Institute of Computer Science, Tel-Aviv University.

TRELEAVEN, P. C., BROWNBRIDGE, D. R., AND HOPKINS, R. P. 1982. Data-driven and demand-driven computer architectures. *Comput. Surv. 14*, 1, 93–143.

TU, H-C. 1988. FAC: Functional array calculator and its application to APL and functional programming. Ph.D. dissertation, Dept. Computer Science, Yale Univ. Available as Res. Rep. YALEU/DCS/RR-468.

TU, H-C., AND PERLIS, A. J. 1986. FAC: A functional APL language. *IEEE Software 3*, 1, 36–45.

TURING, A. M. 1936. On computable numbers with an application to the entscheidungsproblem. *Proc. London Math. Soc. 42*, 230–265.

TURING, A. M. 1937. Computability and λ-definability. *J. Symbolic Logic 2*, 153–163.

TURNER, D. A. 1976. SASL language manual. Tech. Rep. Univ. St. Andrews.

TURNER, D. A. 1979. A new implementation techinque for applicative languages. *Softw. Pract. Exper. 9*, 31–49.

TURNER, D. A. 1981. The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*. ACM, pp. 85–92.

TURNER, D. A., 1982. Recursion equations as a programming language. In *Functional Programming and Its Applications: An Advanced Course*. Cambridge University Press, New York, pp. 1–28.

TURNER, D. A. 1985. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*. Springer-Verlag LNCS 201, pp. 1–16.

VAN HEIJENOORT, J. 1967. *From Frege to Gödel*. Harvard University Press, Cambridge, Mass.

VEGDAHL, S. R. 1984. A survey of proposed architectures for the execution of functional languages. *IEEE Trans. Comput. C-23*, 12, 1050–1071.

VUILLEMIN, J. 1974. Correct and optimal implementations of recursion in a simple programming language. *J. Comput. Syst. Sci. 9*, 3.

WADGE, W. W., AND ASHCROFT, E. A. 1985. *Lucid, the Dataflow Programming Language*. Academic Press, London.

WADLER, P. 1986. A new array operation. In *Workshop on Graph Reduction Techniques*, Springer-Verlag LNCS 279.

WADLER, P. 1987a. Efficient compilation of pattern-matching. In *The Implementation of Functional Programming Languages*, S. L. Peyton Jones, Ed. Prentice-Hall International, Englewood Cliffs, N.J., Chapter 5.

WADLER, P. 1987. Views: A way for pattern-matching to cohabit with data abstraction. Tech. Rep. 34. Programming Methodology Group, Chalmers Univ. of Technology, March 1987. Preliminary version appeared in the *Proceedings of the 14th ACM Symposium on Principles of Programming Languages* (January 1987).

WADLER, P., AND BLOTT, S. 1989. How to make ad hoc polymorphism less ad hoc. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages*. ACM, pp. 60–76.

WADLER, P., AND MILLER, Q. 1988. An Introduction to Orwell. Tech. Rep. Programming Research Group, Oxford University. (First version, 1985.)

WADSWORTH, C. P. 1971. Semantics and pragmatics of the lambda calculus. Ph.D. dissertation, Oxford Univ.

WATSON, P., AND WATSON, I. 1987. Evaluating functional programs on the FLAGSHIP machine. In *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference*. Springer-Verlag LNCS 274, pp. 80–97.

WEGNER, P. 1968. *Programming Languages, Information Structures, and Machine Organization*. McGraw-Hill, New York.

WIKSTRÖM, Å. 1988. *Standard ML*. Prentice-Hall, Englewood Cliffs, N.J.

WISE, D. 1987. Matrix algebra and applicative programming. In *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference*, Springer Verlag LNCS 274, pp. 134–153.

YOUNG, J. 1988. The Semantic Analysis of Functional Programs: Theory and Practice. Ph.D. dissertation, Dept. Computer Science, Yale Univ., 130–142.

Received May 1988; final revision accepted May 1989.