FUNCTIONAL PROGRAMMING

Functional programming languages support the functional programming style. This style emphasizes the computation of values of expressions, and the definition and application of functions as first-class data values. Construction of a program involves composition of individual functions. In contrast to the functional languages, imperative languages, which are the most widely used languages, emphasize the evaluation of statements and have little support for manipulating functions as first-class values. Programs are constructed from individual statements with complex interactions between the statements.

Functional languages are the subject of ongoing research in extending semantics and in implementation. Research into implementations is directed to improving sequential execution and achieving (semi)automatic parallel execution. Sequential implementations have improved dramatically in recent years to the point where performance rivals comparable C programs.

The first section, entitled "Programming with Functions," describes the advantages of functional programming. The next section describes the theory underlying functional programming. The section entitled "History" describes three of the most widely used functional programming languages. Next, strategies and issues in implementing high-performance functional languages are described. Finally, the section entitled "Incorporating State" describes methods for restoring state to functional languages.

PROGRAMMING WITH FUNCTIONS

While many functional programming languages provide facilities for performing assignments and programming as if in an imperative language, much can be gained by programming with functions alone. These advantages are described in the subsections that follow.

Ease of Reasoning

In the presence of assignment statements, variables can change value during execution, reducing the ability of programmers to reason algebraically about their programs. In

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 1999 John Wiley & Sons, Inc.

contrast, functional programs contain equalities which are true for all time. Consider the following program [written in the Haskell language (1)] for finding the roots of a quadratic expression, $ax^2 + bx + c$:

```
quadroot a b c =
  let d = 2*a
    s = sqrt(b*b − 4*a*c) in
  ((−b + s)/d, (−b − s)/d)
```

To prove that this program conforms to the usual formula for computing quadratic roots, substitute the definitions for `d` and `s` into the final expression of the `quadroot` function. This substitutibility property is referred to as *referential transparency.*

Automatic Memory Management

In functional programming, memory allocation and reclamation is performed exclusively by the run-time support system of the language. For example, evaluation of the expression `[1,2,3]` in Haskell creates a list with three elements, prints the contents of the list, and then because there are no other references to this list, the memory used by the list is reclaimed and made available for other lists. By contrast, in imperative languages, when programmers use pointers to allocated data, they must ensure the following issues are resolved.

1. Adequate space for the data must be allocated, and a pointer must be set to this space.
2. The pointer can only be dereferenced after the data have been allocated.
3. When the data are no longer needed, the data should be deallocated and returned to a free memory pool.
4. Pointers to deallocated data should not be dereferenced.

The absence of pointers eliminates these concerns. Automatic memory management has become recognized as a vital security mechanism in Java (2) and other languages for the Internet.

Recursion

Without assignment statements, looping through numeric values or through elements of a data structure is achieved with recursion. For example, the following program (again in Haskell) computes the sum of a list of numbers:

```
sum [] = 0
sum (x:l) = x + (sum l)
```

The alternate cases for the `sum` function are presented on two separate lines. When `sum` is applied to an empty list (whose pattern is `[]`), the function yields a value `0`. Otherwise, `sum` is applied to a nonempty list whose first element is `x`, and the remaining elements are in sub-list `l`. The matching pattern for this case is `x : l`. Function `sum` is called recursively on sub-list `l`. When the sum of sub-list `l` is added to the first element of the list (`x`), the result is the sum of the entire list `x : l`.

To prove that this program is correct, it will be necessary to show that for every list $l = [x_1, \ldots, x_n]$ and number s, sum_aux $s\, l = s + x_1 + \cdots + x_n$. The proof can be conducted inductively over list length. The proof will be described in detail, because it can serve as a prototype for other correctness proofs of recursive functions.

BASIS CASE: Consider the summation of an empty list (whose length is 0). The `sum` function is defined to be 0, which is the sum of the elements of an empty list.

INDUCTION HYPOTHESIS: Suppose that for every list $l' = [x_2, \ldots, x_n]$, sum $l' = x_2 + \cdots + x_n$.

INDUCTIVE CASE: Consider a list $l = [x_1, x_2, \ldots, x_n] = x_1 : l'$. Then:

$$\begin{aligned} \text{sum}\, x_1 : l' &= x_1 + \text{sum}\, l' \\ &= x_1 + (x_2 + \cdots + x_n) \end{aligned}$$

Note that the recursive structure of the proof reflects the recursive structure of the primary data type lists. The proof also relied heavily on referential transparency: the ability to substitute expressions of the programming notation (e.g., sum l') for expressions of the problem domain (e.g., $x_2 + \cdots + x_n$).

Partially Applied Functions

In functional languages, functions that are defined on multiple parameters can be applied to a single argument at a time. When a function of multiple arguments is redefined in this way, it is said to be *curried.* At each argument application, a new function is returned. As an example, consider the following addition function:

```
add x y = x + y
```

The result of the application `add 1` is a new function of a single argument that always adds 1 to its argument.

Higher-Order Functions

Frequently, functions perform similar operations that can be abstracted to a more general function, parameterized with functional arguments. For example, a general reduction function would be defined as follows:

```
red f i [] = i
red f i (x:l) = red f (f i x) l
```

The red function takes three arguments: a binary function, an identity element, and a list. The binary function is to be applied to all elements of the list. Application of red to an empty list results in the identity element.

Now, the `sum` function can be redefined to use the `red` higher-order function as follows:

```
sum = red add 0
```

Here, summation is defined as applying the add function "between" all elements of a list. When applied to an empty list, the identity element `(0)` is returned.

A function that computes the product of all elements of a list can be defined in a similar manner, taking advantage of a similar computational pattern.

```
product = red mul 1
mul x y = x * y
```

Modularity

When creating large software systems, it is often vital to divide the system into smaller discrete modules. The success of this division depends on independence of each module from others, insulating one module from changes in other modules. Independence is achieved in three dimensions: (1) independence in the order in which modules are evaluated; (2) independence in the choice of data structures; (3) independence in the implementation of data structures. Functional languages enhance programmers' ability to achieve these forms of independence.

Imperative programming languages enforce a specific order of evaluation from the statement level to the level of modules. Communication between modules is often performed through construction of, and assignments to, global data structures. Consequently, modules must maintain a rigid time ordering, otherwise, globals will not be correctly initialized by the time they are used. In contrast, functional languages require only that input arguments must be created before a function is invoked. Temporal constraints are relaxed with lazy evaluation, which suspend functions until arguments are sufficiently elaborated. Languages that support curried functions also remove temporal constraints, permitting arguments to functions to be applied incrementally.

Higher-order functions provide both procedural and data abstraction. Procedurally, common activities or patterns can be encapsulated within a single function. For example, a function that evaluates one of a list of other functions depending on a numeric selector may be presented as follows:

```
select :: Int -> [a->b] -> (a->b)
select 0 f:l = f
select (n+1) (f:l) = select n l
```

To abstract this example in the direction of data, lists can be viewed as functions, which return a value stored at a particular numeric index. Consequently, the select function can be abstracted further to have the type `select:: Int -> (Int->a) -> a`.

Finally, functional languages support modular independence by providing separation of implementations of data structures from their external interface. These facilities have been provided by some of the earliest functional languages. Goguen, Thatcher and Wagner (3) realized the relationship between abstract data types, algebras, and equational languages, which are similar to functional languages without higher-order functions.

A function that computes the product of all elements of a list can be defined in a similar manner:

```
product = red mul 1
mul x y = x * y
```

CONCEPTUAL BACKGROUND

Lambda Calculus

Functional programming languages are all based on a much simpler foundation, called *lambda calculus* (4). The simplicity of lambda calculus and its computational rules accounts for the simplicity of functional programming languages, which in turn makes programming far less error-prone. All computable functions can be stated in lambda notation. Lambda notation is specified with the following grammar:

$$
\begin{aligned}
&v \in \text{Variables}\\
&e \in \text{Expressions}\\
&e \quad ::= \quad v \qquad \text{(variable)}\\
&\qquad\quad \lambda v.e_1 \qquad \text{(abstraction)}\\
&\qquad\quad e_1\, e_2 \qquad \text{(application)}
\end{aligned}
$$

While (numeric) constants and primitive operators can be added to this language to improve its readability, this grammar is satisfactory for a discussion of the foundations of the lambda calculus.

A *free* variable in the lambda expression is a variable that occurs outside the scope of any λ-binding. More formally, a variable v is free within lambda expression e in the following cases, based on the form of e:

- When $e = v$.
- When $e = \lambda w.e_1$, $w \neq v$, and v is free within e_1.
- When $e = e_1\ e_2$, and either v is free in e_1 or v is free in e_2.

A *closed* lambda expression has no free variables, and it represents a value, even if the value is a function. A lambda expression with free variables represents a range of values, once all of its free variables are assigned values. The assignment of values is recorded in an environment, which maps syntactic variables to semantic values. If v is a variable and ρ an environment, $\rho(v)$ represents the value assigned to variable v in environment ρ. If, in addition, x is a value, $\rho[v \leftarrow x]$ represents the new environment that satisfies the following property:

$$
\rho[v \leftarrow x](w) = \begin{cases} x & \text{if } v = w \\ \rho(w) & \text{otherwise} \end{cases}
$$

The semantic interpretation of a lambda expression e in an environment ρ is given by a meaning function $\mathscr{M}(e, \rho)$, specified by examining the form of e.

- If e is a variable, $\mathscr{M}(e, \rho) = \rho(e)$.
- If $e = \lambda v.e_1$ is an abstraction, $\mathscr{M}(e, \rho)$ is a function f such that $f(x) = \mathscr{M}(e_1, \rho[v \leftarrow x])$.
- If $e = e_1\ e_2$ is an application expression, $\mathscr{M}(e, \rho) = \mathscr{M}(e_1, \rho)(\mathscr{M}(e_2, \rho))$.

With the meaning of lambda expressions given by meaning function $\mathscr{M}$, computation with lambda expressions is given by a calculus. Soundness and (partial) completeness properties tie the semantics to the calculus. The lambda calculus specifies rules for reducing lambda expressions. The reduction rules rely on a rigorous definition of substitution which respects lambda-bound variables. Given expressions e_1 and e_2 and variable v, $e_1[v/e_2]$ denotes substitution of all free occurrences of v in e_1 with e_2 so that no free variables in e_2 are bound by abstractions in e_1. The rules for performing substitu-

tion are the following:

$$v'[v/e] = \begin{cases} e & \text{if } v = v' \\ v' & \text{otherwise} \end{cases}$$

$$(e_1\, e_2)[v/e] = e_1[v/e]\, e_2[v/e]$$

$$(\lambda w.e_1)[v/e] = \begin{cases} \lambda w.e_1 & \text{if } v = w \\ \lambda w.e_1[v/e] & \text{if } v \neq w, \text{ and } w \text{ not free in } e \\ \lambda w.(e_1[w/w'])[v/e] & \text{if } v \neq w,\ w' \text{ is a new variable, and } w \text{ is free in } e \end{cases}$$

As a consequence of these rules, renaming of variable x occurs in the substitution $(\lambda x.y)[y/(x + 1)]$ prior to replacement, resulting in $\lambda x'.(x + 1)$, and avoiding capture of the free variable x within $x + 1$ by the lambda abstraction.

The lambda calculus is used to determine equality of lambda expressions through reduction. Reduction of an expression e_1 to an expression e_2, denoted $e_1 \rightarrow e_2$, is specified with the following two rules:

β Rule: $(\lambda v.e_1)\, e_2 \rightarrow e_1[v/e_2]$.

η Rule: $\lambda v.e \rightarrow e$ whenever e contains no free occurrences of v.

Rewriting sequences can be composed: $e_1 \rightarrow^* e_2$ if either $e_1 = e_2$ or there exists an expression e' such that $e_1 \rightarrow e'$ and $e' \rightarrow^* e_2$ ($\rightarrow^*$ is the reflexive, transitive closure of the $\rightarrow$ relation).

In addition to the reduction rules, equality of lambda expressions is determined, modulo renaming of lambda-bound variables. As for conventional programming languages, the choice of variable names does not impact the meaning of programs. Equality in the lambda calculus realizes this property by enhancing equality with an α-conversion rule as follows:

$$\begin{aligned} e &\equiv e \\ \lambda v.e &\equiv \lambda w.e[v/w] \\ \lambda v.e &\equiv \lambda v.e', \quad \text{if } e \equiv e' \\ e_1\, e_2 &\equiv e_1'\, e_2', \quad \text{if } e_1 \equiv e_1' \text{ and } e_2 \equiv e_2' \end{aligned}$$

The following theorem guarantees that the order of reductions does not affect the determination of equality.

Theorem 1 (Church–Rosser I) If $e_1 \rightarrow^* e_2$, then there exist expressions e_1' and e_2', where $e_1' \equiv e_2'$ and such that $e_1 \rightarrow^* e_1'$ and $e_2 \rightarrow^* e_2'$.

A normal form of an expression e is an expression e' such that $e \rightarrow^* e'$ and e' cannot be reduced further. As a corollary to the Church–Rosser Theorem, if an expression has a normal form, it is unique (modulo variable renaming). For suppose e_1 and e_2 are normal forms for an expression e, and $e_1 \not\equiv e_2$. Then, according to the Church–Rosser Theorem, there exist expressions e_1', e_2', and e' such that:

- $e_1 \rightarrow^* e_1'$, $e \rightarrow^* e'$, and $e_1' \equiv e'$
- $e_2 \rightarrow^* e_2'$, $e \rightarrow^* e'$, and $e_2' \equiv e'$

Since e_1 and e_2 are normal forms, $e_1 = e_1'$ and $e_2 = e_2'$, and $e_1 \equiv e' \equiv e_2$, contradicting the assumption that e_1 and e_2 are distinct normal forms.

The location within an expression at which a reduction takes place is called a *redex*. Even though the choice of redex does not affect equality (according to the Church–Rosser Theorem), a poor choice of redexes may not result in a normal form. Two rules for locating redexes are commonly used:

Normal Order: The leftmost-outermost redex is always reduced first. When an argument is applied to a function, the function is always evaluated first.

Applicative Order: The rightmost redex is always reduced first. When an argument is applied to a function, the argument is always evaluated first.

To illustrate the problem with applicable order, reduction of the following expression will not terminate:

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

In normal order evaluation, the normal form is $\lambda z.z$, while in applicative order, there is no normal form because each reduction step results in an expression identical to the original.

Theorem 2 (Church–Rosser II) The normal order reduction sequence will always obtain the normal form of a lambda expression (if one exists).

Despite normal order's obvious advantage over applicative order in reaching a normal form, when applicative order is able to find a normal form, the number of applicative order reduction steps is usually less than the number of steps required by a normal order reduction sequence.

HISTORY

The four languages described in this section—Lisp, Scheme, ML, and Haskell—represent significant developments in functional programming languages. They are all widely accepted; and at the same time, they synthesize many ideas current at the time of their development.

Lisp

The earliest functional language was Lisp, developed by J. McCarthy at the end of the 1950s and the beginning of the 1960s at MIT (5). Many of the attributes of Lisp are still present in modern functional languages.

Lisp introduced a language that had simple textual formation rules, automatic memory management, and the concept of a list data structure. Lisp was widely adopted in the Artificial Intelligence community. Its acceptance was due to several features of the language. First, Lisp makes creation and comparison of symbols very easy. For example, the value of an object's color property may be the Lisp symbol `blue`. In contrast to imperative languages, colors do not need to be encoded as integers. All storage allocation and reclamation is performed by the run-time system, so that storage for symbols is managed automatically.

The primary data structure of Lisp is the list, which can have arbitrary form and can therefore represent complicated

relationships between objects. For example, the Lisp list `'(cat isa feline)` represents an "isa" relationship between animal species. Again, storage management by the Lisp run-time system makes it easier to use lists in Lisp than it is to define record structures and explicitly allocate and deallocate storage in imperative languages.

Lisp's facility to use functions as expressions has been retained in modern functional languages. In Lisp, a function is created with a three-element list:

- The first element of the list is the symbol `lambda`, denoting a function-valued expression.
- The second element is a list of parameter symbols.
- The final element is the expression that will be evaluated when the function is applied to arguments.

For example, the following expression contains a two-element list, consisting of two functions—one that increments its argument, and the other that decrements its argument:

```
'((lambda (x) (+ x 1)) (lambda (x) (- x 1)))
```

Lisp/s functions can used like any other value: They can be stored within lists, as in the example above, and can be passed as arguments to other functions. It is for this reason that Lisp's functions are said to be "first-class" values. This facility is well-used within the Artificial Intelligence community. For example, an object's behavior could be dictated during program execution by assigning a particular set of functions to the object.

Though Lisp has a number of aspects that make it a less-than-ideal functional programming language, many of its features have found their way into later functional languages.

Scheme

Following the dissemination of Lisp, the Scheme language (6) was produced to resolve a number of difficulties inherent with Lisp. Scheme solved Lisp's problem of "dynamic scoping" and optimized evaluation of tail-recursive functions. Scheme has roughly the same syntax as Lisp (based on "S-expressions"), but can be evaluated more efficiently. There are two fundamental reasons for Scheme's improved efficiency. First, in Lisp, values of nonlocal variables must be searched for on the run-time stack (later versions of compiled Lisp require declarations of global variables to eliminate searches). Searching for nonlocal variables requires that symbolic names of variables need to be stored on the stack, and accesses to those variables will be slow. The following program displays how dynamic scoping works:

```
(set f (lambda (x) (g)))
(set g (lambda () (+ x 1)))
```

When function `f` is called, it pushes its parameter, `x`, onto the run-time stack. This parameter is available to function `g` when `g` computes `1 + x`.

By contrast, Scheme uses static scoping for variables. Whenever a variable is accessed, its position in the stack can be determined by static analysis of the program. The Scheme compiler will report a syntax error on the program above.

The second reason for Scheme's potential performance advantage over Lisp is that Scheme requires all recursive functions to be tail-recursive. Roughly speaking, in the terminology of Haskell, a function is tail-recursive if all equations defining the function are tail-recursive. An equation is tail-recursive if, whenever the right-hand-side expression is recursive, the recursive call is at the outermost position. The `sum_aux` noted earlier in this article is tail-recursive, and achieves this property by introducing an accumulator. The `sum` function could be written without tail-recursion as follows:

```
sum l = sum_aux 0 l
sum_aux s [] = s
sum_aux s (x:l) = sum_aux (s + x) l
```

Each recursive call of a function that is not tail-recursive will require an additional activation stack frame to be added to the run-time stack to store the function's current state. By contrast, tail-recursive functions can be compiled into iterative loops. As a result, only one run-time activation stack record will be needed for the function call.

When functions are not originally tail-recursive, they need to be transformed into tail-recursive form. Two transformations are generally performed.

Accumulator Parameters. In the first transformation the programmer creates "accumulator parameters" to store intermediate results. For example, in the `sum_aux` function noted earlier in this article, the first parameter holds the accumulated sum.

Continuations. Compilers can automatically transform programs to tail-recursive form by introducing "continuations" (7). A continuation suspends the entire computation of the recursive function until the base case is reached. Consider the following Haskell function definition for appending one list to the beginning of another.

```
append [] l2 = l2
append (x:l1) l2 = (x:(append l1 l2))
```

Function `append` is not tail-recursive. A continuation parameter can be introduced to produce an equivalent (tail-recursive) function definition. Continuations are functions. Lisp's functional form `(lambda (x) e)` is expressed in Haskell with `\x->e`.

The tail-recursive form of the `append` function is the following:

```
appendc l1 l2 = appendcc l1 l2 (\l->l)
appendcc [] l2 c = l2
appendcc (x:l1) l2 c = appendcc l1 l2 (\l->(c
  (x:l)))
```

In both equations for `appendcc`, the continuation parameter (`c`) is a function of a single argument. This argument is assumed to be the result of appending the list arguments to `appendcc`. In the first equation defining `appendcc`, the result of appending the empty list `[]` to `l2` is `l2`, so `c` is applied to `l2`. This result must be applied to the continuation so that suspended computations within the continuation can be invoked. In the second equation defining `appendcc`, a new continuation is created. Its single argument is the assumed result of appending lists `l1` and `l2`. The first element of `x:l1`, which is `x`, is added onto list `l` to produce the result of appending `x:l1` to `l2`. As in the first equation, this result is applied to the continuation in order to invoke suspended computations.

One problem with continuations is that data structures supporting the new continuation functions need to be stored

on the run-time stack. Continuations make heavy use of static scoping, and they can only be used with great care in Lisp.

ML

Following the development of Lisp by over a decade, ML became a functional language for a broader user community (8). ML provides type inference, polymorphism, and pattern-matching. These features are present in most later functional programming languages. In the early 1990s, the specification of ML was standardized, resulting in the new language, SML (9). SML has a wide variety of implementations, compiling either to bytecodes or directly to machine language. In this section, all examples will be written in Haskell's syntax to maintain uniformity of notation. Translation of Haskell to ML is a straightforward detail.

Within imperative programming languages, types are assigned to variables primarily to allocate storage. Types do not mandate a range of values for variables because variables are assigned to memory locations, and memory can be altered with arbitrary values. By contrast, in functional languages, every variable and function can be assigned a fixed domain, which is a countable, partially ordered set of possible values. The ML language compiler is able to infer denotations of the domains of all variables and functions at compile time. The simple structure of the language makes the inference rules relatively simple to specify. Type inference is able to catch semantic programming errors and at the same time ease the programming task.

As an example of type inference, consider the following function definition:

```
inc x = x + 1
```

The definition above declares a function, `inc`, of a single parameter, `x`. Function `inc` will correctly compute a result only if it is applied to a number. Furthermore, the only nonerror results are numeric. These observations are inferred by the ML compilation system, which issues the following type of judgment:

```
inc :: Int -> Int
```

ML also provides pattern-matching within function definitions. As an example, consider the following function definition:

```
map f [] = []
map f (x:l) = (f x):(map f l)
```

The `map` function in this definition applies a function `f` to every element of a list. When `map f` is applied to an empty list (denoted with `[]`), the empty list is returned. When `map f` is applied to a nonempty list, the second definition of `map` applies. Variable `x` will then be bound to the head of the list, and variable `l` will be bound to the remaining elements of the list. Following these bindings of values to variables, the right-hand-side expression will be evaluated, creating a list whose first element is the result of applying `x` to function `f` and whose remaining elements are formed from recursively applying the `map` function to `f` and list `l`. Notice that the operator in the pattern `x:l` is identical to the list construction operator ":" in the expression's right-hand side.

The final innovation introduced by ML is provision for polymorphic functions within the type inference system. Since ML can use functions as arguments to other functions, it is natural to provide for functions to be used in general contexts. Consider the `map` function presented above. This function applies its argument to every element of a list. The function is equally valid when it is applied to lists of integers, characters, and so on. ML provides a notation in its type system for describing such polymorphic functions. Function `map` has the following type denotation:

```
map :: (a -> b) -> [a] 0> [b]
```

Within this type expression, `a` and `b` stand for unique universally quantified variables spanning the domain of all (variable-free) type denotations. Type expressions `[a]` and `[b]` stand for lists containing elements of types `a` and `b`, respectively.

Haskell

Haskell was developed by a committee of researchers in an effort to standardize lazy functional languages. Haskell has many similarities with Miranda (10). The Hope language (11) preceded Miranda, but originally only performed lazy evaluation on constructors. Both Lisp and ML are *strict* functional languages: When applying an argument to a function, the argument is fully evaluated before the function's code is evaluated. This argument-passing mechanism is called *call-by-value,* and it realizes applicative-order evaluation. It is used in a number of conventional programming languages. However, there are several reasons why this mechanism is somewhat undesirable.

1. When a parameter's value is not used within a function, computation of its value is unnecessary.
2. When a parameter's value is not used within a function, and computation of its value causes an error condition to occur, unnecessary error conditions may arise.
3. The presence of call-by-value in a programming language causes increased complexity in the language's definition, because conditionals and streams (unbounded-length lists) must be defined to be part of the language.

Haskell and other *lazy* functional languages evaluate expressions only when they are needed. Furthermore, because every expression has only one value throughout the entire execution of a program, once an argument is evaluated, its value overwrites the expression.

Lazy evaluation in Haskell resolves the three problems identified above:

1. Given the following Haskell function definition, evaluation of `f -1 x` will not evaluate `x`.

   ```
   f i x = if i < 0 then 0 else x
   ```

2. Had the expression `f -1 (1/0)` been evaluated with the function definition above, an error would not be reported because the second argument is not needed.
3. Lazy evaluation is able to supplant a number of seemingly essential features found in strict functional languages. For example, the conditional expression `(if ...then ...else)` can be defined in Haskell itself (albeit with less syntactic clarity).

```
cond True t f = t
cond False t f = f
```

More complicated control structures can also be created without adding additional facilities to the language.

ML has a *stream* extension that is already a natural part of Haskell. In Haskell, the components of a data structure are only evaluated if the components' values are demanded. As a result, the computation of infinite lists and other data structures can be specified, while only evaluating a finite portion. In addition, if a function is applied to all elements of a data structure, the function will only be computed on those elements that are actually demanded.

Programming with streams enables programmers to use lists as tables of functions. For example, the following function produces a list of (non-negative, integral) powers of a number `x`.

```
powers x = 1 : (map (\t->x*y) (powers x))
```

If the first three elements of the `powers x` stream are demanded, evaluation proceeds in the following steps:

```
powers x
-> 1 : (map (\y->x*y) (powers x))
-> 1 : (map (\y->x*y) (1 : (map (\y->x*y)
                                (powers x))))
-> 1 : ((\y->x*y) 1) : (map (\y->x*y)
                            (map (\y->x*y)
                                 (powers x)))
-> 1 : x*1 : (map (\y->x*y)
                  (map (\y->x*y) (powers x)))
-> 1 : x*1 : (map (\y->x*y)
                  (map (\y->x*y)
                       (1 : (map (\y->x*y)
                                 (powers x)))))
-> 1 : x*1 : (map (\y->x*y)
                  ((\y->x*y) 1) : (map (\y->x*y)
                                       (map (\y->x*y)
                                       (powers x))))
-> 1 : x*1 : (map (\y->x*y)
                  (x*1 : (map (\y->x*y)
                              (map (\y->x*y)
                              (powers x)))))
-> 1 : x*1 : ((\y->x*y) x*1) : (map (\y->x*y)
                                    (map (\y->x*y)
                                    (map (\y->x*y)
                                    (powers x)))))
-> 1 : x*1 : x*x*1 : (map (\y->x*y)
                          (map (\y->x*y)
                               (map (\y->x*y)
                               (powers x))))
...
```

The first three members of this stream contain the values of `x0`, `x1`, and `x2`. The evaluation steps displayed above were chosen using normal-order evaluation. This reduction rule is guaranteed to find a normal form if one exists.

One problem that is evident in the reduction steps displayed above is that values of the expression `powers x` are continually recomputed. Since Haskell disallows the side effects present in imperative programs, `powers x` will be unchanged whenever it is invoked. To save the computed values of a function, Haskell implements *full laziness:* When an expression is reduced to a value, the value overwrites the expression. In the preceding steps, subexpression `powers x` will be rewritten in two steps as depicted in Fig. 1. In this figure, the nodes labeled : construct lists. The nodes labeled @ apply arguments to functions. Depiction of full laziness shows that results of computations are reused.

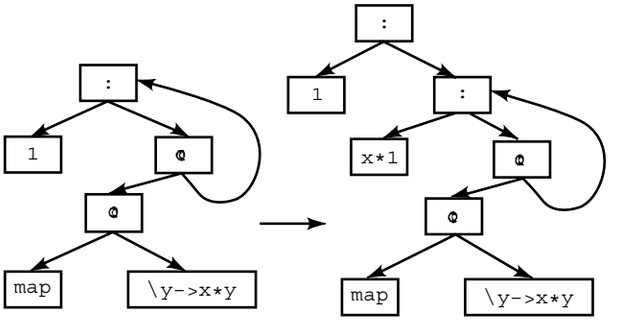

Figure 1. Evaluation of `powers x` with full laziness.

Haskell exploits lazy evaluation in a number of useful ways. Lists can be specified as a generator and filter with the *list comprehension* syntax. for example, the stream of powers of a number `x` can be specified as the expression `[x^n | n <- [0..]]`. This generator is a lazy stream producer, equivalent to the Haskell expression `nats 0`, where `nats` is defined as follows

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

In addition, lazy evaluation implements a form of "memoization." In essence a table is maintained that maps expressions to previously computed values. When a new value is to be computed (`powers x`, for example), any previously computed value is returned.

IMPLEMENTATIONS

McCarthy's paper describing Lisp contained an interpreter written in Lisp (5). Its implementation calls for provision of the primitive list-handling and arithmetic functions and could serve as the specification of a Lisp interpreter, written in another language.

Common to all implementations of functional languages, representations are needed for expressions whose evaluation has been suspended. Lisp's dynamic binding removes the need for special representations. Some implementations represent expressions as graphs, as portrayed in Fig. 1. Other statically scoped functional languages require creation of special data structures (12).

Closures

Execution of strict functional languages requires creation of data structures representing partially applied functions, which can be passed as arguments to other functions. A *closure* is a pointer to a function's code, along with a list of activation stack frames binding nonlocal variables to values. Each binding of a value v to a variable `x` will be expressed as $x \mapsto v$. A closure whose function is f = \y->e with bindings $\beta = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$ will be expressed as $\langle\langle f, \beta \rangle\rangle$.

To see how closures are created, consider the following function definitions:

```
add x y = x + y
inc = add 1
```

When evaluating the expression `add 1`, the closure ⩽`y->x+y,[x ↦1]`⩾ will be created. This closure can be assigned to a symbol with the definition noted earlier in this article.

When the `inc` function is applied to a value, say `5`, a new binding for `y` will be pushed onto the closure's activation stack. The resulting activation stack will have the following bindings: `[{y ↦ 5}, {x ↦ 1}]`. Within this environment, the expression `x+y` will be evaluated, returning the value `6`.

Thunks

Lazy functional languages delay evaluation of expressions until they are demanded. When demanded, delayed expressions need to be evaluated under bindings for variables that were in place at the time they were originally delayed. A *thunk* is a data structure containing an expression and an activation stack. The thunk suspending expression e under binding list β will be expressed as $\lhd e, \beta \rhd$.

In the `powers` function noted in the previous section, the following data structure is constructed on evaluation of the expression `powers 2`:

```
1 : ◁map(\y->x am y)(powers x), {[x ↦ w]}▷
```

The thunk carries enough information to determine the value of `x` when the lambda expression `\y->x * y` and the application expression `powers x` are ultimately used.

Creation of closures and thunks constitute serious challenges to obtaining performance comparable to conventional programming languages. To maximize performance of functional programs, various compilation strategies have been proposed. These strategies usually assume the target machine executes an instruction set tailored to functional languages. Realization of the compiler is usually performed in two steps. First, the functional program is compiled into the idealized target language. Next, the target language is either (a) interpreted or (b) translated further to the target language of an actual computer. Several target functional instruction sets have been proposed. Of these, the SECD and CAML instruction sets will be described in more detail.

SECD

The SECD machine is one of the earliest abstract functional programming language engines. It was described by Landin in 1964 (13). The machine has four sections:

- **Stack:** holds intermediate results during expression evaluation.
- **Environment:** holds the current list of bindings of variables to values.
- **Code:** holds a list of SECD machine instructions to execute.
- **Dump:** stores environments that are unneeded when making function calls.

Each of these four sections is represented as a list. Occasionally, closures will be created and stored on the stack. Elements of the dump are triples, each consisting of a stack, environment, and code list. Implicitly, there is a storage area for lists, closures, and triples, called the *heap*. Data structures stored in the heap have reference tags. Memory allocated to structures without outstanding references are placed in a free area for later reuse.

The instruction set of the machine consists of the following instructions.

- `const(n)`: Loads a number `n` onto the stack.
- `vari(i)`: Loads the `i`th item in the environment onto the stack.
- `add`: Adds the top two stack elements and pushes the result onto the stack. In an actual implementation there would be additional primitive operations.
- `lambda(`c`)`: Creates a closure out of the new code list c and the current environment and then pushes this structure onto the stack.
- `apply`: The stack must consist of a value `x` and a closure $\langle\langle c, e\rangle\rangle$. The code list c is evaluated in environment e, after its extension with x. Before evaluation of code list c, the dump is extended with a triple consisting of the current stack, the current environment, and the remainder of the code list.

When the machine reaches the end of a code list, if the dump is empty, the machine stops, with the result of the computation at the top of the stack. Otherwise, the dump is popped and its topmost triple used to restart the computation that was suspended by a previous `apply` instruction.

Compilation of a simple functional language into instructions of the SECD machine will be described next. The functional language's grammar is described as follows:

$n \in$ Numeric constants
$v \in$ Variables
$e \in$ Expressions

e	$::=$	n	(constant)
		v	(variable)
		$e_1 + e_2$	(primitive operation)
		$\backslash v - > e_1$	(abstraction)
		$e_1\, e_2$	(application)

A compilation function C is defined to map each expression and a description of the environment to a code list. The compilation rules are described as follows:

$C(n,\ \rho)$	$=$	`[const(`n`)]`
$C(v_i, [v_1, \ldots, v_i, \ldots, v_n])$	$=$	`[var(`i`)]`
$C(e_1 + e_2,\ \rho)$	$=$	$C(e_1,\ \rho) +\!\!+ C(e_2,\ \rho) +\!\!+$ `[add]`
$C(\backslash v - > e,\ \rho)$	$=$	`[lambda(`$C(e,\ (x : \rho))$`)]`
$C(e_1\, e_2,\ \rho)$	$=$	$C(e_1,\ \rho) +\!\!+ C(e_2,\ \rho) +\!\!+$ `[apply]`

In the rules listed above, the $+\!\!+$ operator joins lists of elements. For example, in the rule for compiling addition expressions of the form $e_1 + e_2$, three lists are concatenated:

1. The result of compiling e_1: $C(e_1, \rho)$.
2. The result of compiling e_2: $C(e_2, \rho)$.
3. The add instruction: `[add]`.

As an example of the SECD machine, consider compilation of the function application `(\x->x + 1) 5`. The compilation

steps are the following:

$C((\backslash x{-}{>}\, x+1)\,5,\,[])$
$= C(\backslash x{-}{>}\, x+1,\,[]) +\!\!+ C(5,\,[]) +\!\!+ [\texttt{apply}]$
$= [\texttt{lambda}(C(x+1,\,[x]))] +\!\!+ [\texttt{const(5), apply}]$
$= [\texttt{lambda}(C(x,\,[x]) +\!\!+ C(1,\,[x]) +\!\!+ [\texttt{add}]),\ \texttt{const(5), apply}]$
$= [\texttt{lambda}([\texttt{var(1), const(1), add}]),\ \texttt{const(5), apply}]$

Letting c = `[var(1), const(1), add]`, execution of the code list derived above proceeds in the following steps. Each step describes a state of the SECD machine, with each of its parts specified as a list.

S	E	C	D
[]	[]	[lambda(c) const (5), apply]	[]
[⟨⟨c, []⟩⟩]	[]	[const(5), apply]	[]
[5, ⟨⟨c, []⟩⟩]	[]	[apply]	[]
[]	[5]	c	[([5, ⟨⟨c, []⟩⟩], [], [])[
[5]	[5]	[const(1), add]	[([5, ⟨⟨c, []⟩⟩], [], [])[
[1, 5]	[5]	[add]	[([5, ⟨⟨c, []⟩⟩], [], [])[
[6]	[5]	[] []	[([5, ⟨⟨c, []⟩⟩], [], [])[
[6, 5, ⟨⟨c, []⟩⟩]	[]	[]	[]

The complexity of the SECD machine, along with the complexity of function application and return, has led researchers to explore simpler architectures. For example, the FPM machine, (Ref. 10, Chap. 15) consists of a single stack (and an implicit heap).

CAM

The *Categorical Abstract Machine* (CAM) (14) combines the implementation ideas of the graph reduction machines and the SECD machine (and its successors). In the CAM, programs are transformed to combinator expressions. A combinator is lambda expression without free (unbound) variables. In the CAM, each of the fixed set of combinators is evaluated when all arguments are supplied. Each combinator then acts as a rule that transforms the order of its arguments. For example, the S combinator is defined by the following rewrite rule:

$$S\,x\,y\,z = x\,z\,(y\,z)$$

As a lambda expression, S would be defined as follows:

$$S = \backslash x{-}{>}\, \backslash y{-}{>}\, \backslash z{-}{>}\, x\,z\,(y\,z)$$

The advantage of retaining combinator expressions in the combinator form is that simple rewriting rules can be attached to each combinator.

A set of *categorical combinators* are first defined axiomatically. In the axioms, environments represent bindings of variables to values. Environments are extended when arguments are applied to functions. An environment x is extended with an argument y by forming a pair (x, y). These axioms are the axioms of "weak categorical combinatory logic" and are listed in Table 1.

To convert lambda expressions to expressions involving the categorical combinators, the expressions are first converted to *De Bruijn* notation. Every lambda expression $\backslash v{-}{>}e$ is replaced by the De Bruijn expression $\lambda(e')$, where every occurrence of variable v in e is replaced by a variable marker of the form $\#i$, with i denoting the nesting depth of v in the environment. With expressions in De Bruijn notation, a compilation function C is defined as follows:

Constants: $C(c) = {'}c$
Built-ins: $C(+) = \Lambda(+ \circ \mathrm{Snd})$
Variables: $C(\#i) = \mathrm{Snd} \circ \mathrm{Fst}^i$
Pairs: $C((e_1, e_2)) = \langle C(e_1), C(e_2)\rangle$
Application: $C(e_1\ e_2) = \mathrm{App} \circ \langle C(e_1), C(e_2)\rangle$
Lambda: $C(\lambda(e)) = \Lambda(C(e))$

For example, the lambda expression $(\backslash x{-}{>}x\ +\ 1)\ 5$, which increments 5, is expressed in De Bruijn notation as $\lambda(+(\#0,\ 1))\ 5$. The De Bruijn expression compiles to the following combinator expression:

$$\mathrm{App} \circ \langle \Lambda(\mathrm{App} \circ \langle \Lambda(+ \circ \mathrm{Snd}),\ \langle \mathrm{Snd},\ {'}1\rangle\rangle),\ {'}5\rangle \qquad (1)$$

To create a reduction sequence, the combinator expression is applied to the empty environment, denoted $()$. A strict reduction sequence proceeds as follows:

$$\begin{aligned}
& \mathrm{App} \circ \langle \Lambda(\mathrm{App} \circ \langle \Lambda(+ \circ \mathrm{Snd}),\ \langle \mathrm{Snd},\ {'}1\rangle\rangle),\ {'}5\rangle() \\
\xrightarrow{\text{assoc}}\ & \mathrm{App}(\langle \Lambda(\mathrm{App} \circ \langle \Lambda(+ \circ \mathrm{Snd}),\ \langle \mathrm{Snd},{'}1\rangle\rangle),{'}5\rangle()) \\
\xrightarrow{\text{depair}}\ & \mathrm{App}(\Lambda(\mathrm{App} \circ \langle \Lambda(+ \circ \mathrm{Snd}),\ \langle \mathrm{Snd},\ {'}1\rangle\rangle)(),\ {'}5()) \\
\xrightarrow{\text{quote}}\ & \mathrm{App}(\Lambda(\mathrm{App} \circ \langle \Lambda(+ \circ \mathrm{Snd}),\ \langle \mathrm{Snd},\ {'}1\rangle\rangle)(),\ 5) \\
\xrightarrow{\text{ac}}\ & \mathrm{App} \circ \langle \Lambda(+ \circ \mathrm{Snd}),\ \langle \mathrm{Snd},\ {'}1\rangle\rangle((),\ 5) \\
\xrightarrow{\text{assoc}}\ & \mathrm{App}(\langle \Lambda(+ \circ \mathrm{Snd}),\ \langle \mathrm{Snd},\ {'}1\rangle\rangle((),\ 5)) \\
\xrightarrow{\text{depair}}\ & \mathrm{App}(\Lambda(+ \circ \mathrm{Snd}((),\ 5),\ \langle \mathrm{Snd},\ {'}1\rangle((),\ 5)) \\
\xrightarrow{\text{depair}}\ & \mathrm{App}(\Lambda(+ \circ \mathrm{Snd})((),\ 5),\ (\mathrm{Snd}((),\ 5),\ {'}1((),\ 5)) \\
\xrightarrow{\substack{\text{snd,}\\ \text{quote}}}\ & \mathrm{App}(\Lambda(+ \circ \mathrm{Snd})((),\ 5),\ (5,\ 1)) \\
\xrightarrow{\text{ac}}\ & + \circ \mathrm{Snd}(((),\ 5),\ (5,\ 1)) \\
\xrightarrow{\text{assoc}}\ & +(\mathrm{Snd}(((),\ 5),\ (5,\ 1)) \\
\xrightarrow{\text{snd}}\ & +(5,\ 1) \\
\longrightarrow\ & 6
\end{aligned}$$

Table 1. Axioms of Weak Categorical Combinatory Logic

quote:	${'}cx$	$= c$
fst:	$\mathrm{Fst}(x, y)$	$= x$
snd:	$\mathrm{Snd}(x, y)$	$= y$
depair:	$\langle f, g\rangle\ x$	$= (fx, gx)$
ac:	$\mathrm{App}(\Lambda(f)x, y)$	$= f(x, y)$
assoc:	$(f \circ g)\ x$	$= f(gx)$

The axioms of the categorical combinators could be directly implemented by an interpreter, or a graph-reduction machine. The CAM goes further, by breaking each axiom into smaller steps. The CAM comprises three sections:

Term: a pair describing the environment.

Code: a list of instructions to execute.

Stack: intermediate results held for temporary storage.

CAM instructions are defined for each combinator symbol. The CAM instructions are defined below:

fst: This instruction is produced from the Fst combinator. If the term is a pair (x, y), term x is pushed onto the stack.

snd: This instruction is produced from the Snd combinator. If the term is a pair (x, y), term y is pushed onto the stack.

push: This instruction is produced from the $<$ element of the pairing combinator. The term is pushed onto the stack.

swap: This instruction is produced from the intermediate (,) element of the pairing combinator. The term and top of stock are swapped.

cons: If the current term is y and the top of stack is x, the new term is (x, y), and the stack is popped.

curry(c): This instruction is produced from the combinator expression $\Lambda(c)$. If x is the current term, the new term will be closure $\langle\langle c, x\rangle\rangle$.

app: This instruction is produced from the App combinator. The current term must be the pair $(\langle\langle c, x\rangle\rangle, y)$. Instruction sequence c will be executed with new term (x, y).

quote(x): The new term will be x.

add: If the current term is (x, y), the new term will be $x + y$.

In creating an instruction sequence from an expression $e_1 \circ e_2$, the sequence for e_1 will be appended to the end of the sequence for e_2. The combinator expression [Eq. (1)] produces the following three code sequences, with execution proceeding from s0:

s0: push; curry(s1); swap; quote(5); cons; app;

s1: push; curry(s2); swap; push; snd; swap; quote(1); cons; cons; app;

s2: snd; add

While the set of combinators in the CAM are fixed, another approach is to define a set of combinators "on-the-fly," depending on the expressions present in a program. The resulting rewriting sequences are often shorter than the sequences produced by the CAM.

Dataflow

Due to referential transparency, every expression (without free variables) in a functional program represents a single value. This property suggests a unique method of execution, called *dataflow* (15). In the dataflow model every expression in a functional program is associated with a dataflow instruction, which consists of the following fields:

Operation Code: This field is filled in by the compiler, and it may consist of a primitive arithmetic function, or one of a collection of dataflow operators such as `merge`, `apply`, and `switch`.

Inputs: This field is an array of slots that are filled during execution with the inputs to the operation.

Destination: This field is filled in by the compiler; and it contains the address of the instruction, and slot in its input list, that will hold the result of the current operation.

Dataflow machines (in both hardware and software realizations) have an *execution queue,* which holds all instructions that have sufficient operands to be executed. Execution of an instruction proceeds in the following steps:

1. Get an instruction from the execution queue.
2. Evaluate the instruction, obtaining result r to be sent to instruction i at input slot s.
3. Copy r into slot s of instruction i.
4. If sufficient slots of s have been filled, add i to the execution queue.

Dataflow programs are usually depicted in dataflow diagrams, which are directed graphs with nodes representing instructions. Dataflow diagrams are interpreted like Petri nets. Each edge may pass a token, representing the result of an instruction to a slot of another instruction. For example, the `quadroot` program can be represented as the dataflow diagram of Fig. 2.

While all instructions of the program perform according to the execution outline presented above, all instructions have unique behavior. Below is a list of the instructions appearing in the `quadroot` program, as well as others needed to support conditionals, and function calls.

copy: This instruction propagates the value at its input to multiple outputs.

constant: This instruction sends a single constant value to its output without waiting for inputs.

pair: This instruction forms a data structure in "I-structure" memory consisting of both input values, and it returns a pointer to the new structure.

switch: This instruction implements conditionals. The instruction sends the input at its second input slot to its output if the value on its first input is the Boolean `T`; otherwise, the input at its third input slot is passed to the output.

apply: This instruction applies arguments to a function pointer.

The `pair` instruction creates a structure in a section of memory called the *I-structure memory.* I-structures have behavior similar to the idea of lazy evaluation. When applied to an I-structure pair, selectors `fst` and `snd` do not return values until the appropriate slots in the I-structure receive values. In effect, each slot of an I-structure has a queue of pending requests.

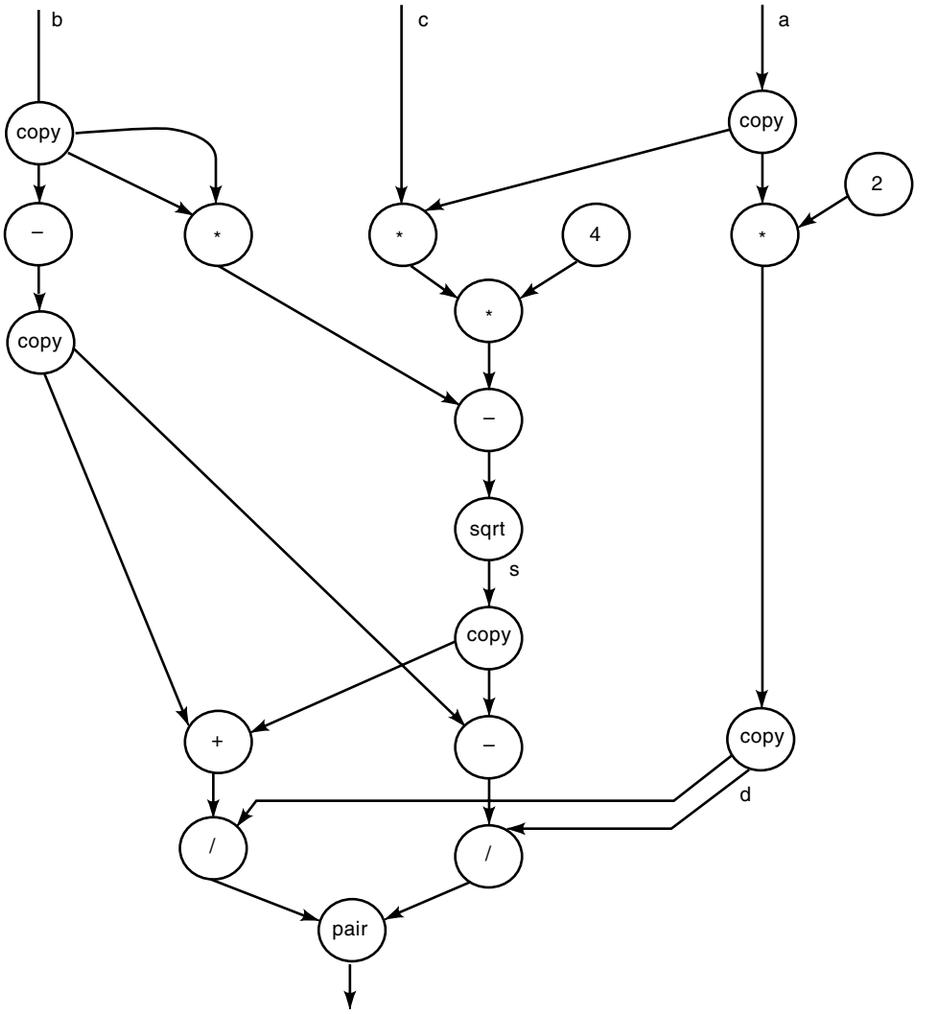

Figure 2. Dataflow diagram for `quadroot`.

The `apply` instruction creates a new instance of a function's code, and it sends the arguments to `apply` to the instantiated function. To perform function instantiation efficiently, instructions are tagged with an invocation number when they are added to the execution queue. When a value is computed by an instruction, the value is copied to the input slot of the destination instruction with the identical invocation number.

In data-driven evaluation, many opportunities for parallelism exist. So much parallelism is present that applications require strategies to pursue parallel execution in only the most critical instructions, rejecting other less critical opportunities. In several large-scale applications, execution of programs written in the dataflow language SISAL compares favorably with execution of Fortran programs (16).

Other proposals for massively-parallel evaluation of functional programming languages have also been advanced. Parallel graph reduction performs the kind of graph evaluation described in Fig. 1 at multiple points of a graph simultaneously (17). The `Nesl` language (18) contains arrays as a fundamental data type so that element accesses can be performed in a "data-parallel" manner in constant time. Various parallel primitive operations, and the higher-order parallel `map` operation are applied to arrays. Programs tend to be written in a divide-and-conquer style, to achieve logarithmic-time performance. In contrast with dataflow programs, parallelism is under the control of the programmer. But the programmer is not responsible for explicitly creating and controlling parallel processes.

INCORPORATING STATE

The state of a program is an encapsulation of its inputs, outputs, and memory. The computation of a conventional language program depends critically on the state. Due to referential transparency, however, evaluation of a functional program is invariant with respect to the state. Without state, though, functional languages must explicitly pass all inputs to functions, and cannot update variables "in place."

As an example of the inefficiency introduced without in-place update, consider the `incall` function defined below, which increments all values of a list.

```
incall [] = []
incall (n:l) = (n+1):(incall l)
```

A naive implementation essentially copies the entire list as it traverses down to its end. If the list is a component of another

data structure, copying will avoid the side effect of changing the value of the data structure. However, if the list is contained in no other data structure, copying is an unnecessary expense.

Two approaches have been advanced to make it possible for functional programming languages to recognize situations where in-place update can be performed. The first approach requires program analysis to recognize situations where values are referenced from a single point in the program. The second approach has programmers employ abstract data types that can be updated in place while retaining functional semantics.

Single Threadedness

A data structure is *single-threaded* if it is referred to directly from at most one other data structure (19). If a functional language compiler can determine that all data structure arguments to a function call are single-threaded, a variant of the function performing destructive updates can be called. A data structure is *multithreaded* if it is not single-threaded.

To determine if a variable always holds single-threaded values, the compiler infers which parameters to each function are definitely single-threaded. Because the problem of deciding whether a variable is single-threaded is undecidable, a conservative inference procedure must be employed. For each function with n parameters, the inference procedure outlined below will deduce an n-tuple of Boolean values, where the ith value is `T` if the ith parameter is judged to be single-threaded. The procedure will also deduce if a function returns single-threaded results.

1. Initially, for every function defined in the program, all arguments are assumed to be single-threaded, so each function is assigned an n-tuple consisting only of `T` values.
2. Suppose the body of a function f contains a variable x. If x is a formal parameter of the function and is multithreaded (the corresponding element of f's tuple is `F`), all uses of x will be multithreaded. Also, if there are multiple occurrences of x within f, all uses will be multithreaded. If x is the ith parameter to a function g and the use of x is multithreaded, the ith element of g's tuple is set to `F`. The analysis is continued on g.
3. Suppose the body of function f contains multiple occurrences of a single-threaded parameter, or at least one occurrence of a multithreaded parameter. Then the function returns multithreaded results.
4. Suppose the body of a function contains a call to a function f whose ith argument is gx. If g returns a result that is multithreaded, the ith parameter of f is multithreaded (the ith element in f's n-tuple will be `F`). The analysis is continued on f.

The analysis outlined above will eventually terminate because, in the worst case, all functions will be associated with n-tuples of values that are entirely `F`, and once an element is set to `F` it cannot be revised.

In the previous example, if function `inclist` is invoked with the expression `inclist (inclist [0..10])`, `inclist` will be inferred to be single-threaded. On the other hand, `inclist` is multithreaded in the expression `(\l->append (inclist l) (inclist l)) [0..10]`. The analysis will find the parameter of `inclist` to be multithreaded, since `l` occurs twice in the calling expression. Therefore, the result of `inclist` is multithreaded, and `append` is multithreaded.

Monads

Monads have been introduced to incorporate a notion of state within functional languages. Monads can be thought of as abstract data types that are internally single-threaded.

While monads are usually treated as a concept of category theory, Wadler (20) described them as a generalization of the "list comprehension," which is present within the Haskell language. A functional language implementation of state transformers can perform destructive assignments without violating referential transparency.

One important use of monads has been to implement input-output in version 1.4 of the Haskell language. Input-output operations typically produce side-effects. For example, reading from a file returns the next object in the file, but also advances the file's position so that the next read gets the next object. In Haskell monads encapsulate the side-effects of input-output, so that functions using the special `IO` monad are not exposed to side-effects.

SUMMARY

Functional languages ensure referential transparency, where every expression describes a single value (once free variables are bound to values). Functional languages can be described denotationally, without use of a store. In addition, they can be defined axiomatically with Cartesian closed categories. As a result, reasoning can be conducted equationally directly in the notation of the language.

The earliest functional language, Lisp, gave way to a variant, Scheme, which relies on static scoping. ML provides more conventional syntax, and more easily supports currying. While ML (and Scheme) employ strict evaluation, Haskell evaluates function arguments with lazy evaluation. As a result, it is possible to program with streams of unbounded length and make use of other unbounded data structures.

Functional languages have been used primarily in Artificial Intelligence applications, due to the heavy use of recursively defined data structures. Reasoning about types has also been extremely important in these applications, and its influence has been felt in other areas of programming language design.

While functional languages have been criticized for poor performance, due mainly to the absence of side effects, several proposals have been advanced to incorporate side effects within the evaluation system, retaining referential transparency at the language level.

BIBLIOGRAPHY

1. P. Hudak et al., Report on the functional programming language Haskell, Version 1.2, *ACM SIGPLAN Notices,* **27** (5): 1992.
2. J. Gosling, B. Joy, and G. Steele, *The Java™ Language Specification,* Reading, MA: Addison-Wesley, 1996.
3. J. A. Goguen, J. W. Thatcher, and E. G. Wagner, An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, *Current Trends in Programming*

Methodology (Yeh, ed.), Chap. 5, Englewood Cliffs, NJ: Prentice-Hall, 1978.

4. D. Scott, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory,* Cambridge, MA: MIT Press, 1977.
5. J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, *Commun. ACM,* 3(4), 184–195, 1960.
6. G. J. Sussman and G. L. Steele, Jr., *Scheme: An interpreter for an extended lambda calculus,* MIT AI Memo No. 349, 1975.
7. O. Danvy and A. Filinski, Abstracting control, *1990 ACM Conf. Lisp Funct. Program.,* 1990, pp. 151–160.
8. M. J. Gordon, A. J. Milner, and C. P. Wadsworth, Edinburgh LCF, *Lect. Notes Comput. Sci.,* **78**: 1979.
9. R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML,* Cambridge, MA: MIT Press, 1989.
10. D. Turner, An overview of Miranda, *ACM SIGPLAN Notices,* **21** (12): 158–166, 1986.
11. A. J. Field and P. G. Harrison, *Functional Programming,* Reading, MA: Addison-Wesley, 1988.
12. S. N. Kamin, *Programming Languages: An Interpreter-Based Approach,* Reading MA: Addison-Wesley, 1990.
13. P. J. Landin, The mechanical evaluation of expressions, *Comput. J.,* **6**: 308–320, 1964.
14. M. Mauny and A. Suarez, Implementing functional languages in the categorical abstract machine, *1986 ACM Conf. Lisp Funct. Program.,* 1986.
15. Arvind and R.S. Nikhil, Executing a program on the MIT tagged-token dataflow architecture, *1987 SEAS Spring Meet.,* 1987, pp. 1–29.
16. J. T. Feo, D. C. Cann, and R. R. Oldehoeft, A report on the Sisal language project, *J. Parallel Distrib. Comput.,* **10**: 349–366, 1990.
17. S. L. Peyton Jones, Parallel implementations of functional programming languages, *Comp. J.* **32** (2): 175–186, 1989.
18. G. E. Blelloch et al., Implementation of a portable nested data-parallel language, *J. Parallel Distrib. Comput.,* **21**: 4–14, 1994.
19. P. Hudak, A semantic model of reference counting and its abstraction, *1990 ACM Conf. Lisp Funct. Program.,* 1986, pp. 351–363.
20. P. Wadler, Comprehending monads, *1990 ACM Conf. Lisp Funct. Program.,* 1990, pp. 61–78.

CLIFFORD WALINSKY
The Portland Group, Inc.

FUNCTIONAL MATERIALS. See FUNCTIONAL AND SMART MATERIALS.