A Survey of Sequential and Parallel Implementation Techniques for Functional Programming Languages

Ralph C. Hilzer, Jr. Lawrence A. Crowl

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

Technical Report 95-60-05

May 1995

Abstract

This paper surveys sequential and parallel implementation techniques for functional programming languages, as well as optimizations that can improve their performance. Sequential implementations have evolved from simple interpreters to sophisticated super-combinator-based compilers, while most parallel implementations have explored a broad range of techniques. We analyze the purpose and function of each implementation technique and discuss the current state-of-the-art in functional language implementation.

Contents

1 Introduction **2**
- 1.1 Functional Programming 2
- 1.2 Referential Transparency 3
- 1.3 Language Variants 4
- 1.4 Lambda Calculus 6

2 Common Sequential Evaluation Methods **9**
- 2.1 Abstract Syntax Tree Interpreters 10
- 2.2 SECD Machine 13
- 2.3 Fixed-Combinators 16
- 2.4 Graph Reduction of Lambda Expressions 17
- 2.5 Super-Combinators 22
- 2.6 Compilation 25

3 Optimizations **30**
- 3.1 Tail Recursion Elimination . . . 31
- 3.2 Memoization 32
- 3.3 λ-Argument Copying 35
- 3.4 Partial Evaluation 37
- 3.5 Mixed-Order Evaluation 38

4 Parallel Implementation Techniques **41**
- 4.1 Invoking Parallelism 42
- 4.2 Conservative and Speculative Parallelism 44
- 4.3 Memory Organization 48
- 4.4 Task Blocking and Resumption . 50
- 4.5 Task and Data Distribution . . . 53
- 4.6 Task Grain Size 58
- 4.7 Garbage Collection 60
- 4.8 Aggregate Memory Access 61
- 4.9 Vectorization 63

5 Summary **63**

A Introduction to Haskell **68**

1 Introduction

This paper surveys the current state of both sequential and parallel functional programming language implementations. First, background information on functional programming (§ 1.1), referential transparency (§ 1.2), language variants (§ 1.3), and lambda calculus (§ 1.4) is presented to provide context for the survey. If background unneeded, the remainder of the introduction may be safely bypassed.

The sequential section (§ 2) describes several environment-based and combinator evaluation methods, each with roots in graph theory. A common example illustrates the operation of each method and serves as a basis for comparing them.

The optimizations section (§ 3) presents a number of methods to improve sequential performance such as: moving up delayed computations, providing more sharing, eliminating unnecessary copying, precomputation, and using mixed (lazy and eager) evaluation.

The parallel section (§ 4) assumes readers are aware that each processor of a parallel system employs an evaluation method described in the sequential section and may also incorporate improvements from the optimizations section. Consequently, other issues are addressed such as what the general categories of parallel systems are, how parallel tasks are invoked and controlled, memory organizations that support parallel execution, task blocking and resumption, the placement and balancing of tasks and data in memory, the average size of parallel tasks, garbage collection, accessing data as aggregates, and vectored operations. Actual characteristics of several implementations are included so that readers may judge the state of current parallel implementations in these categories for themselves.

The summary (§ 5) brings major points of the paper together and attempts to make sense out of them. The picture that emerges reflects a significant amount of progress and budding opportunity for improvement in future systems.

Wherever possible, code segments in this paper are presented in Haskell. Appendix A presents a brief introduction to Haskell for those who are unfamiliar with this highly expressive functional language.

1.1 Functional Programming

Functional programming was introduced by John McCarthy and others [1962] as an alternative to the more conventional *imperative* programming style used by FORTRAN and COBOL. Since then, functional programming has been refined and improved by a number of others, many of whom drew from longstanding and well-established research into computable functions by mathematicians such as Schönfinkel [1924] and Church [1941].

Functional or *applicative* programming languages belong to a general class of languages in which the underlying model of computation is *function application* [Church, 1941; Schönfinkel, 1924]. Functions map a set of objects called the *domain* into objects in a "target set" called the *codomain* or *range*. Function application invokes that mapping, assigning actual arguments in the domain (supplied by a function call) to formal parameters in the function definition, completing an expression-name association. Once the associations are complete, operations on formal parameters in the function body can be used to compute a result or set of results in the function range.

Imperative languages also employ functions. Although many other features are often used to distinguish functional and imperative languages, a fundamental difference is in their treatment of variables. Whereas functional languages only associate variable names with expressions (or values once the expressions are evaluated), imperative languages associate variable names (other than functions) with memory locations [Abelson *et al.*, 1985; Böhm *et al.*, 1991]. Functional variables never change value, but imperative variables do.

Functions appeal to mathematicians because they are very expressive and incorporate many useful mathematical properties such as suitability for theorem proving. Similarly, functional languages appeal to some programmers because use of the math-like syntax and semantics exports many mathematical properties to the programming language. For example, proof of program correctness can be reasonably straightforward with functional programs, whereas it is usually not with imperative programs.

Furthermore, concurrent (multi-processor) programming can also be simpler with functional languages than with imperative languages. Imperative program components often contain dependencies that require synchronization during execution, whereas executable functional program components are independent and therefore are free of dependencies.

Favorable mathematical properties and concurrency potential make functional programming languages an active and productive research topic. Paul Hudak [1989] chronicled this research from the perspective of a language lexicon, highlighting the distinguishing features and specific contributions of several languages. This paper extends that survey to include a number of sequential and parallel functional-language implementation techniques. Readers unfamiliar with functional languages are encouraged to consult the Hudak article. It is easy to read and very informative.

1.2 Referential Transparency

A language in which the evaluation of program expressions does not introduce side effects is said to be *referentially transparent* [Backus, 1978]. In referentially transparent programs, the whole is determined by the sum of the parts, so neither the position of those parts nor the substitution of parts with different but equivalent parts can change the program's outcome. Therefore, the parts are like black boxes. They require initial testing, but no further adjustment is necessary if used properly.

It is the association of variable names with values that makes functional languages referentially transparent. Imperative languages are *not* referentially transparent because, when variable names are associated with memory locations, the values may change. When values are allowed to change, the result produced by any section of code may depend on when it is executed. This type of dependency is not possible in referentially transparent languages. Some implications of referential transparency are:

Static Semantics The semantics of functional languages are closer to traditional mathematical notation than are the semantics of imperative languages [Appleby, 1991]. In particular, the semantics are static—independent of time.

This static property makes it possible to analyze and determine the behavior of functional program code statically, whereas dynamic analysis is necessary with imperative languages. Static (versus dynamic) characteristics can also make it easier to describe the semantics of a language using an abstract notation such as operational, axiomatic, or denotational semantics [Field and Harrison, 1988; Backus, 1978], and makes it possible to prove the correctness of programs (including non-trivial programs) using a technique similar to mathematical induction.

Scheduling Flexibility As referentially transparent expressions become eligible for evaluation, the order used to evaluate them can neither change their values nor alter the program results. Therefore, schedulers can be very flexible. They are free to select from among several task alternatives choosing the best one for the current set of circumstances. In parallel systems they can assign the tasks to multiple processors without fear of interference caused by side effects.

Single-Assignment Variables The association of variable names with values has a consequence that those values cannot change over the lifetime of the variable. For that reason, functional-language variables are often called *single-assignment variables*.

A negative side of single-assignment variables is that they do not allow imperative-style assignment statements. Without the assignment statement, repetition with iterative looping is not possible. Other techniques such as recursion must be used instead. Unfortunately, recursion generally consumes more space and time than iteration.

1.3 Language Variants

The expressive features used in functional languages vary widely from language to language. Some of the more common language variants are:

First- and Higher-Order Functions *First-order* functional languages allow either all or part of a language data set (excluding functions) to have the status of first-class data types, meaning they can be passed as parameters or returned as function values. In *higher-order* functional languages, functions themselves are elevated to the status of first-class data types so that they too can be passed as parameters or returned as function values [Abelson *et al.*, 1985].

Nearly all functional languages include some form of higher-order functions. The exceptions usually involve experimental techniques where the point is more clear using first-order functions [Takano, 1991]. Unfortunately, higher-order functions also increase the complexity of both the language and the language interpreter. Some languages, such as FP, restrict higher-order functions to a small fixed set called *combining forms* or *functional forms*, reasoning that the increased power gained when more choices are included only leads to chaos [Backus, 1978].

Polymorphism Functions that are polymorphic can accept arguments of varying data types. Functions with *parametric polymorphism* are indifferent to data types of their arguments [Hudak, 1989]. They behave identically regardless of the argument type. For example, in the identity function, `ID`, below:

$$\texttt{ID x = x} \qquad \forall\, \texttt{x}$$

`x` is parametrically polymorphic. `ID` returns `x` no matter what type of `x` is input.

The behavior of functions with *ad hoc polymorphism* or *overloading* does depend on the types of arguments that are input. For example, an overloaded function might perform integer add when integer arguments are input and floating-point add when floating-point arguments are input.

Similar to higher-order functions, polymorphism improves language flexibility and expressibility. Polymorphism also reduces the need to rewrite sections of code. This can make programs shorter and more concise. Therefore, program design and readability may benefit.

For example, consider the simplicity of a polymorphic program that uses one routine to sort a number of different data types (e.g., integer, real, and alphabetic) as opposed to a non-polymorphic program that must have a different sorting routine for each data type.

Unfortunately, polymorphism has some negative features. For example, consider that some non-polymorphic imperative languages are able to detect and report data type errors at compile time. Since polymorphic data types may not be assigned until execution time, error checking may be delayed until execution time as well.

Imperative Features Some languages such as LISP, ML, and Scheme allow imperative features such as the imperative-style assignment statement (§ 1.2) to support looping and iteration [McCarthy *et al.*, 1962; Gordon *et al.*, 1979; Steel and Sussman, 1975]. These features are intended to improve performance by increasing speed and/or reducing storage requirements. They also make functional languages more palatable to imperative programmers. Unfortunately, they destroy referential transparency, so "pure" functional languages normally either do not include them or discourage their use. SISAL is an exception, allowing imperative features in `for` statements and externally invoked functions. In both cases, when execution of the imperative code is complete, only value a single value is returned to the calling function, so referential transparency is preserved [Böhm *et al.*, 1991].

Eager and Lazy Evaluation The function $\mathtt{f}\ \mathtt{x}_1\ \mathtt{x}_2\ \ldots\ \mathtt{x}_n$ where $n \geq 0$, is said to be *strict* on argument $\mathtt{x}_i$, where $1 \leq i \leq n$, if it always requires the value of $\mathtt{x}_i$ for evaluation; otherwise it is *non-strict* on $\mathtt{x}_i$ [Field and Harrison, 1988; Peyton Jones, 1987]. For example, the function:

```
g x y = x + y
```

is strict on both arguments x and y since both are always required to evaluate $+$. Conversely, the function:

```
h x y z = if x>10 then y else z      (1)
```

is strict on `x`, but non-strict on `y` and `z` (`x` is always needed to evaluate `h`, but `y` is only needed if $\mathtt{x} > \mathtt{10}$, and `z` is only needed if $\mathtt{x} \leq \mathtt{10}$.

Functional languages are said to use *eager evaluation* if they evaluate all arguments before the function application, regardless of whether those arguments are strict or non-strict. For example, eager evaluation would evaluate all arguments in function `h` above. If an argument is not needed (e.g., argument `b` in the call `(h 12 a b)`, its evaluation represents unnecessary work. If the argument is not needed and fails to terminate (e.g., it is infinitely recursive or involves evaluation of an infinite list), the program will also fail to terminate, possibly unnecessarily.

Many early functional programming language implementations used strictly eager evaluation including FP, ML, an early version of Hope, and some data flow languages [Backus, 1978; Gordon *et al.*, 1979; Burstall *et al.*, 1980; Hudak, 1989].

Functional-language implementations are said to use *lazy evaluation* if evaluation of arguments is delayed until after application of the containing function. Lazy arguments are evaluated only when they are needed. Unneeded arguments are never evaluated. If each instance of a shared argument is evaluated separately, the implementation is said to be *partially lazy*. Conversely, if shared arguments are evaluated only once, the implementation is said to be *fully lazy*.

To illustrate the difference between lazy and eager evaluation, assume $\perp$ is an expression whose evaluation fails to terminate. Notice that in equation 1 above $(\mathtt{h}\ \mathtt{12}\ \mathtt{10}\ \perp)$ terminates normally returning 10 using lazy evaluation, but fails to terminate using eager evaluation. Consequently, some programs that terminate with lazy evaluation, fail to terminate with eager evaluation.

If evaluation of an expression terminates using both lazy and eager evaluation, both methods will return the same results, although sometimes in a quite different manner. For example, eager evaluation would execute slower and consume more space than lazy evaluation if unneeded arguments were frequently encountered. On the other hand, eager evaluation is easier to implement. Therefore eager evaluation can execute faster if early evaluation of strict arguments makes their values more conveniently available than lazy evaluation's late evaluation.

Another important difference between lazy and eager evaluation is in how they handle input/output. For example, assume that `ttyin` is a function that returns a list of characters typed at a terminal, and `ttyout` is a function that receives characters and displays them on the terminal screen. Then, the function `(ttyout ttyin)` displays characters typed at the terminal. Notice that eager evaluation requires that `ttyin` complete before any output appears on the screen, whereas lazy evaluation outputs each character to the screen as it is typed [Eisenbach, 1987].

This ability to handle sequences of data with undetermined length (frequently referred to as *streams*) illustrates of one of lazy evaluation's chief attractions. It can do wonders for language expressiveness.

SASL, KRC, Lazy ML, Miranda, Orwell, Ponder, and Haskell are all lazy functional languages [Peyton Jones, 1987].

1.4 Lambda Calculus

The *lambda calculus* is a mathematical calculus for computable functions proposed by Alonzo Church [1941]. It is a simple means of describing the properties of computable functions, effectively treating the functions as rules. Only a few constructs and simple semantics are required. Furthermore, it is expressive, sufficiently so that it can express not only all functional languages but all computable functions as well [Field and Harrison, 1988; Peyton Jones, 1987].

The subsections below describe the structure of lambda expressions (§ 1.4.1), lists definitions associated with these lambda expressions (§ 1.4.2), lists rules to modify lambda expressions structure (§ 1.4.3), and describes lambda lifting, a transformation method that can enhance the suitability of lambda expressions for compilation (§ 1.4.4).

1.4.1 Lambda Expressions

Lambda expressions are unnamed functions consisting of a lambda (λ), some formal parameters, a function body, and the actual arguments [Church, 1941; Field and Harrison, 1988; Peyton Jones, 1987]. For example, in the expression:

$$(\lambda x. + x\ x)\ 4$$

- λ identifies the function as a lambda expression,
- the x prior to the period identifies x as the formal parameter,
- $+\ x\ x$ is the body of the λ-expression, and
- 4 is the actual argument.

The lambda expression above is evaluated (*reduced*) as follows:

$$(\lambda x. + x\ x)\ 4 \quad \rightarrow \ + 4\ 4 \quad \rightarrow \quad 8$$

and a lambda expression with two parameters is reduced in a similar manner:

$$\begin{array}{ll} & (\lambda x, y. * y\ x)\ 3\ 7 \\ \equiv & ((\lambda x.\lambda y. * y\ x)\ 3)\ 7 \\ \rightarrow & (\lambda y. * y\ 3)\ 7 \\ \rightarrow & * 7\ 3 \\ \rightarrow & 21 \end{array}$$

Although this basic form is quite adequate from an expressibility point of view, frequently additional forms are added to provide a simpler representation for more complex expressions.

For example the designers of Miranda include *let* and *letrec* forms in their lambda calculus and refer to it as *enriched* lambda calculus [Peyton Jones, 1987].

Let expressions have the structure:

`let` *formal parameter* = *argument*
`in` *function body*

and are equivalent to lambda expressions. For example,

`let x = 4 in + x x` $\equiv (\lambda x. + x\ x)\ 4$

is a let expression and its equivalent lambda expression. One advantage of the let expression is that it requires no special keyboard characters. Some feel that the meaning of a let expression is clearer than that of an equivalent standard (unenriched) lambda expression.

Letrecs have the same structure as let expressions, but also allow recursion. For example:

```
letrec factorial =
λn.if n == 0 then 1
       else (* n factorial (- n 1))
in (factorial 10)
```

is a letrec expression for `factorial` invoked with an argument of 10. The usefulness of enriched lambda expressions becomes clear when you consider how complex `factorial` would be if it were implemented with standard lambda expressions.

1.4.2 Lambda Definitions

The following definitions are frequently used in conjunction with λ-expressions.

Reducible Expression A functional call is reducible when all of its arguments are available (the arguments may be either reduced or unreduced). A reducible function call is also called a reducible expression or *redex*.

Normal Forms An expression is in *Normal Form* (NF) if it has either been reduced to a unique value, such as 23, or it is a unique data object that cannot be reduced further, such as the list (23, 72, 49, 84).

An expression is in *Head Normal Form* (HNF) if it is either in NF or it is a λ-expression or built-in function with one or more of its outermost arguments (furthest from the function) missing and either there are either no innermost arguments or they are all evaluated to NF. For example, $(+\ 20)$ is in HNF; $(+\ (*\ 5\ 4))$ is not (because $(*\ 5\ 4)$ is an unevaluated argument of $+$.

An expression is in *Weak-Head Normal Form* (WHNF) if it is either in NF, HNF, or it is a λ-expression or a built-in function with one or more of its outermost arguments missing and either evaluated or unevaluated innermost arguments. For example, all of the HNF examples above are in WHNF. The function $(+\ (*\ 5\ 4))$ is in WHNF but is not in HNF.

Bound and Free Variables Variables in the body of a lambda expression can appear either *bound* or *free*. A variable is bound if it is a formal parameter in an enclosing lambda expression. Otherwise, the variable is free. For example, the variable y is bound in the expression:

$$(\lambda y. + y\ x)$$

but the variable x is free.

Instantiation Sometimes lambda expressions are *instantiated*, meaning one copy (or instance) of a common sub-expression is created that can be shared. The instance is simply a graph sub-component. Sharing components have pointers to the instance. One benefit of instantiation is that once the instance is evaluated, other accesses to it do not require re-evaluation.

Instantiation of expressions containing no free variables is easy because, once evaluated, the value of the instantiated expression cannot change. Instantiation of expressions containing free variables is more difficult, because bindings of the free variables can change. However, it is possible when some mechanism is used to maintain current bindings for the free variables. These free-variable bindings are called the *environment* of the expression.

Currying It is possible to treat a function of n arguments as a concatenation of n single-argument functions [Field and Harrison, 1988; Peyton Jones, 1987]. This idea, called *currying*, was proposed by Schönfinkel [1924] but didn't get its name until it was extensively investigated by Curry and Feys [1958].

The function `f` in `(f x y z)` can be interpreted as a function with three arguments `x`, `y`, and `z` or, using currying, as a function with the single argument `x`. That is, currying would interpret the expression as `(((f x) y) z)` or:

- the function `f` with the single argument `x`,

- the function `(f x)` with the single argument `y`, and

- the function `((f x) y)` with the single argument `z`.

For example, currying interprets $(+\ 1)$ as a function requiring one argument. It is equivalent to $(\lambda x. + x\ 1)$. Both add one to their argument.

In languages like Hope and Haskell, functions have multiple arguments, whereas in languages like Miranda they have single arguments (use currying) [Field and Harrison, 1988].

Applicative-Order Reduction *Applicative-Order Reduction* (AOR) reduces the leftmost innermost redex first The leftmost redex is the one whose function operator is textually to the left of all other redexes in the expression, and the innermost redex contains no other redexes. For example, given the function `(f (g arg))`, AOR would reduce the argument `arg` first, then the argument `(g arg)`, and then the function `(f (g arg))`. Since AOR reduces all arguments prior to applying the outermost function, it implements eager evaluation.

Normal-Order Reduction *Normal-Order Reduction* (NOR) reduces the leftmost outermost redex first. The leftmost redex is the one whose function operator is textually to the left of all other redexes in the expression, and the outermost redex is the one that is not contained in any other redex. Given the function `(f (g arg))`, NOR would initiate reduction of the function `f` first, then `f`'s argument `(g arg)` if needed, and finally `g`'s argument `arg` if it is needed. NOR always invokes the function prior to reducing function arguments. The sharing of common sub-expressions (introduced above in instantiation) and NOR implements fully lazy evaluation.

1.4.3 Lambda Rules

Lambda rules are applied to lambda expressions in order to change their form, ultimately with the intent of reducing them (i.e. obtaining a simple answer). The rules fall into the following four categories:

reductions—reduces the expression to a simpler form,

abstractions—abstracts the expression to a more complex form (such as by adding formal parameters), and

conversions—changes the expression to an equivalent form (e.g., through a formal parameter name change).

lifting–removes free variables from an expression so it can be instantiated.

The following rules can be applied to lambda expressions.

Beta-Reduction A *beta* or β-reduction reduces a lambda expression by substituting actual arguments for formal parameters in the body of the lambda expression. For example:

$$(\lambda x. + x\ x)\ 4 \xrightarrow{\beta_r} +\ 4\ 4$$

is a β-reduction.

Beta-Abstraction A *beta* or β-abstraction is the inverse of a β-reduction, abstracting the expression instead of reducing it. For example:

$$+\ 4\ 4 \xrightarrow{\beta_a} (\lambda y. + y\ y)\ 4$$

is a β-abstraction. Note that the name of the formal parameter need not be the same as was used in the β-reduction.

Delta-Reduction A *delta* or δ-reduction reduces a basic function. For example,

$$+\ 4\ 4 \xrightarrow{\delta} 8$$

is a δ-reduction applied to the basic function operator $+$.

Alpha-Conversion An *alpha* or α-conversion changes the name function parameters. For example:

$$(\lambda x. + x\ x) \overset{\alpha}{\leftrightarrow} (\lambda y. + y\ y)$$

is an α-conversion between the formal parameter names x and y. α-conversions are used to avoid name clashes during reduction.

Eta-Conversion An *eta* or η-conversion is used to convert an expression in one representation to another equivalent representation. For example:

$$(\lambda x.\mathsf{f}\ x) \overset{\eta}{\leftrightarrow} \mathsf{f}$$

is an η-conversion where f is a function and x does not occur free in f (§ 1.4.2).

1.4.4 Lambda Lifting

Since y occurs free in the body of the expression:

$$\lambda x. + x\ y$$

the expression cannot be instantiated (and therefore cannot be shared) as is because the value of y depends on the binding it is assigned by enclosing lambda expressions. There are two approaches to instantiating an expression with free variables. First, enclosing environments can added to the expression until all free variables are removed.

Alternatively, a process called lambda lifting can be used. Lambda lifting involves applying β-abstractions and other conversions to remove free variables. The basic idea is to convert an expression with free variables into a function with a formal parameter corresponding to that parameter. For example, the β-abstraction and α-conversion below convert the expression to a new form where all variables in the body of $\lambda\mathtt{w}$ are bound:

$$\begin{array}{ll} & (\lambda x. + x\ y) \\ \overset{\beta}{\rightarrow} & (\lambda y.\lambda x. + x\ y)\ y \\ \overset{\alpha}{\rightarrow} & (\lambda w.\lambda x. + x\ w)\ y \end{array}$$

Therefore, the λw body *can* be instantiated.

2 Common Sequential Evaluation Methods

This section presents common methods that functional programming language implementations use to evaluate programs.

The success of sequential implementations is heavily dependent on their ability to provide good performance, and good performance is closely tied to efficient storage management and fast execution speed. Major contributors to poor performance in functional implementations are:

- Unnecessary reevaluation of redundant expressions,
- A reliance on recursion to perform repetition, and
- Interpreting the code rather than compiling it.

Lazy evaluation can improve both storage management and execution speed by fostering sharing, so one criteria the methods are judged by is how efficiently they perform lazy evaluation. Generally, the early methods in this section require heavy administrative support to provide lazy evaluation whereas the latter methods efficiently implement lazy evaluation.

Unfortunately, lazy evaluation does not provide for sharing among all redundant expressions, so other measures described later (§ 3) are necessary to locate and remove these redundancies.

Another contributor poor performance is a reliance on recursion to perform repetition. Although the evaluation methods presented in this section can do little to correct recursion deficiencies, some optimization techniques described later (§ 3) do address the problem by either improving the efficiency of recursion or moving away from recursion to other repetition methods (of course trying to do so without undermining the functional model).

Finally, although there are arguments to support using both interpretation and compilation to translate programs, if the code is to be executed numerous times, compilation is the preferred technique because it requires less execution-time analysis.

The evaluation methods contained in this section are Abstract Syntax Tree Interpreters (§ 2.1), the SECD Machine (§ 2.2), fixed-combinators (§ 2.3), graph reduction of lambda expressions (§ 2.4), supercombinators (§ 2.5), and compilation methods (§ 2.6).

2.1 Abstract Syntax Tree Interpreters

This subsection describes and analyzes tree-traversal implementations called Abstract Syntax Tree (AST) interpreters. Most early functional-language interpreters were of this type translating program source code to lambda expressions and then, in turn, converting the lambda expressions to a tree format where tree-traversal algorithms are used to reduce them. The advantage of tree-traversal implementations is simplicity. Eager implementations (§ 1.3) are quite straightforward. However, considerable administration must be added to support fully lazy evaluation, so lazy implementations may not be so simple. Furthermore, neither the eager nor the lazy implementations are fast enough to compete with imperative language implementations.

Abstract Syntax Tree An *Abstract Syntax Tree* (AST) is a parse tree where some nodes are annotated (decorated) with semantic actions (meaning) [Field and Harrison, 1988, Chapter 8]. The semantic operators typically include:

- app—apply the function valued expression at the left child node of app to the expression at the right child node,
- int—the child is an integer constant,
- var—the child is a variable reference,
- prim–the child is a primitive function,
- lambda—the left child is a formal parameter; the right child is a lambda body,
- closure—a composite structure consisting of a lambda expression and a list of its current variable bindings.

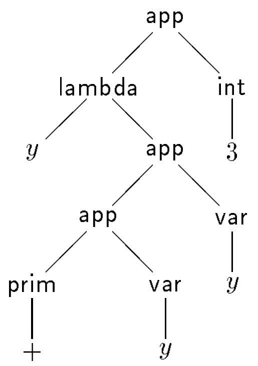


Figure 1: AST Graph for $(\lambda y. + y\ y)\ 3$

In the first phases of interpretation, the program's source code is translated to lambda expressions, and then the lambda expressions are converted to an AST format. AST format for $(\lambda y. +\ y\ y)\ 3$ is:

app(lambda(y, app(app(prim+, var y), var y)), int 3)

The AST format is presented graphically in figure 1.

Eager Interpreter Normally, two functions are used to interpret AST code. They are Apply and Eval [Field and Harrison, 1988, Chapter 9]. Apply returns the WHNF (§ 1.4.2) result of a function applied to its argument. Eval evaluates an expression in a manner specified by a list of predefined rules.

Initially, Eval is applied to the program AST. This and subsequent applications of Eval and Apply transform program components to new reduced forms. Ultimately, the program is reduced to a normal form representing the program result.

Translation rules for Eval, Apply, and some other operators that comprise a small eager interpreter are presented in figure 2. This interpreter leaves the body of a lambda expression intact throughout β-reduction (§ 1.4.3), maintaining variable bindings in a separate structure called the *environment* $[env]$. The β-reduction of each enclosing lambda expression adds a binding (i.e., formal parameter, actual argument) to this environment. When an expression has been reduced to a state where only the lambda body and its environment remain, the lambda body can be evaluated. At that time, bindings for the lambda body variables are accessed from the environment.

The addition of a binding to an environment is represented symbolically by $(v, expr) :: [env]$, where $[env]$ is an environment and $(v, expr)$ is the binding being added (i.e., v is the variable name and $expr$ is the expression bound to it). Accessing a binding from an environment is represented symbolically by $[env] \;\|\; v$ where $[env]$ is the environment and v is the variable name being accessed. If 2 is the most recent binding of v added to $[env]$, then $[env] \;\|\; v$ returns 2.

Evaluation of the expression $(\lambda y. + y\ y)\ 3$ is shown in Figure 3, where each reduction step arrow is annotated with the rule that was applied from figure 2.

Notice that the application of rule 1 passes the current environment to two sub-expressions. That way, each sub-expression can modify and access its own environment. This requires that the implementation support the dynamic creation, access, and termination of environments. Languages such as LISP provide the support as a list of associations.

Notice also that rule 6 creates a composite structure consisting of the lambda expression and its environment. This structure is called a *closure* (depicted by the semantic operator closure in figure 3). Rule 8 uses this closure to add the binding (y, int 3) to the environment. Further down, two applications of rule 3 use this binding to replace instances of var y with int 3 in the lambda body. Each var y consults its own environment to form the binding.

Rule 9 gathers the primitive function's parameters into a list. Prior to the application of rule 9 each argument must be reduced to the extent that it can be evaluated. Rules 12 and 13 perform delta reductions.

Lazy Interpreter In order to convert the eager interpreter of figure 2 to a lazy interpreter it is necessary to add *suspensions* that either temporarily or permanently remove sub-expressions from evaluation consideration. This can be done by changing rule 1 in figure 2 to:

$$\begin{array}{l} \mathsf{Eval}\ (\mathsf{app}\ (E1,\ E2),\ [env]) \quad \Longrightarrow \\ \mathsf{Apply}(\mathsf{Eval}(E1,\ [env]),\ \mathsf{susp}\ (E2,\ [env])) \end{array}$$

If $E1$ is a lambda expression and $E2$ is its argument, this causes the β-reduction to be completed with an unevaluated (suspended) argument. If it is not reawakened, the suspended argument, called a *suspension*, is ignored. Of course, sometimes the suspension *must* be reawakened, so the new rule:

$$\begin{array}{l} \mathsf{Eval}\ (\mathsf{susp}\ (E,\ [env])) \quad \Longrightarrow \\ \mathsf{Eval}\ (E,\ [env]) \end{array}$$

must be added. Furthermore, Eval must be applied to susp (...) in order to invoke this rule. This is accomplished by revising rules 12 and 13 in figure 2 as follows:

$$\begin{array}{l} \mathsf{funct} + (E1,\ E2) \qquad \Longrightarrow \\ \mathsf{int}\ (\mathsf{Eval}\ (E1) + \mathsf{Eval}\ (E2)) \end{array}$$

$$\begin{array}{l} \mathsf{funct} - (E1,\ E2) \qquad \Longrightarrow \\ \mathsf{int}\ (\mathsf{Eval}\ (E1) - \mathsf{Eval}\ (E2)) \end{array}$$

Notice that in this case argument evaluation commences only after evaluation of the containing function has been initiated, or during delta-reduction. Therefore, only required arguments are awakened. Unneeded arguments

1	Eval (app $(E1,\ E2)$, $[env]$)	$\Longrightarrow$	Apply (Eval $(E1,\ [env])$, Eval $(E2, [env])$)
2	Eval (int i, $[env]$)	$\Longrightarrow$	int i
3	Eval (var v, $[env]$)	$\Longrightarrow$	$[env] \parallel v$ (Draw value of v from $[env]$)
4	Eval (prim (p), $[env]$)	$\Longrightarrow$	op $(p,$ arityof $(p),\ nil)$
5	Eval (op (...), $[env]$)	$\Longrightarrow$	op (...)
6	Eval (lambda (...), $[env]$)	$\Longrightarrow$	closure (lambda (...), $[env]$)
7	Eval (closure (...), $[env]$)	$\Longrightarrow$	closure (...)
8	Apply (closure (lambda $(v,\ B)$, $[env]$), A)	$\Longrightarrow$	Eval $(B,\ (v,\ A) :: [env])$
9	Apply (op $(p,\ PAC,\ args)$, A)	$\Longrightarrow$	if $PAC = 1$ then funct $p\ (A :: args)$ else op $(p,\ PAC-1,\ (A :: args))$
10	arityof $(+)$	$\Longrightarrow$	2
11	arityof $(-)$	$\Longrightarrow$	2
12	funct $+$ (int i_1, int i_2)	$\Longrightarrow$	int $(i_1 + i_2)$
13	funct $-$ (int i_1, int i_2)	$\Longrightarrow$	int $(i_1 - i_2)$

Figure 2: Rules For An Eager AST Interpreter

Note: Underlined items show effects caused by application of the previous rule.

$$\text{Eval(app(lambda}(y, \text{app(app(prim }(+), \text{var } y), \text{var } y)), \text{int } 3), [nil])$$

$$\xrightarrow{1} \underline{\text{Apply}}(\underline{\text{Eval}}(\text{lambda}(y, \text{app(app(prim }(+), \text{var } y), \text{var } y)), [nil]), \underline{\text{Eval}}(\text{int } 3, [nil]))$$

$$\xrightarrow{6} \text{Apply}(\underline{\text{closure}}(\text{lambda}(y, \text{app(app(prim }(+), \text{var } y), \text{var } y)), [nil]), \text{Eval}(\text{int } 3, [nil]))$$

$$\xrightarrow{2} \text{Apply(closure(lambda}(y, \text{app(app(prim }(+), \text{var } y), \text{var } y)), [nil]), \underline{\text{int } 3})$$

$$\xrightarrow{8} \text{Eval}\underline{(\text{app(app(prim }(+), \text{var y}), \text{var y}), [(\text{y}, \text{int } 3)])}$$

$$\xrightarrow{1} \underline{\text{Apply}}(\underline{\text{Eval}}(\text{app(prim }(+), \text{var } y), [(y, \text{int } 3)]), \underline{\text{Eval}}(\text{var } y, [(y, \text{int } 3)]))$$

$$\xrightarrow{3} \text{Apply(Eval(app(prim }(+), \text{var } y), [(y, \text{int } 3)]), \underline{\text{int } 3})$$

$$\xrightarrow{1} \text{Apply}(\underline{\text{Apply}}(\underline{\text{Eval}}(\text{prim }(+), [(y, \text{int } 3)]), \underline{\text{Eval}}(\text{var } y, [(y, \text{int } 3)])), \text{int } 3)$$

$$\xrightarrow{4} \text{Apply(Apply}\underline{(\text{op} +, \text{arityof } (+), nil)}, \text{Eval}(\text{var } y, [(y, \text{int } 3)])), \text{int } 3)$$

$$\xrightarrow{10} \text{Apply(Apply(op }(+, \underline{2}, nil), \text{Eval}(\text{var } y, [(y, \text{int } 3)])), \text{int } 3)$$

$$\xrightarrow{3} \text{Apply(Apply(op }(+, 2, nil), \underline{\text{int } 3}), \text{int } 3)$$

$$\xrightarrow{9} \text{Apply(op} +, \underline{1, \text{int } 3}), \text{int } 3)$$

$$\xrightarrow{9} \underline{\text{funct} + (\text{int } 3, \text{int } 3)}$$

$$\xrightarrow{12} \underline{\text{int } (3+3)} \equiv \text{int } 6$$

Figure 3: Eager AST Interpretation of $(\lambda y. + y\ y)\ 3)$

are ignored. This implements partial lazy evaluation.

In order to be *fully lazy* (§ 1.3), shared expressions must be evaluated only once. The lazy AST interpreter described above fails to do this. For example, it would beta reduce $(\lambda y. + y\ y)\ arg$ to $(+\ arg\ arg)$, and then evaluate arg twice during delta-reduction.

Added administration is needed to detect common expressions and avoid their redundant evaluation. For example, multiple instances of formal parameters in lambda bodies (such as the two instances of y in the example above) can be detected and marked during beta reduction, and then, during delta-reduction, the function can evaluate one suspension and apply its result at the positions of other suspensions. Unfortunately this becomes quite complex and cumbersome when it is applied to all shared expressions.

2.2 SECD Machine

The poor performance of AST interpreters is due in large part to the overhead required to maintain an execution environment. Although, SECD is also an environment-based system, it streamlines the administrative organization by using four stacks called the Object **S**tack, **E**nvironment, **C**ontrol, and **D**ump (see [Landin, 1964; Henderson, 1980] or [Field and Harrison, 1988, Chapter 10]). The stacks are controlled by a driver function that changes their contents.

Initially, the control stack contains the input string, and the other stacks are empty. Then, translation rules are applied to change stack contents as follows:

The control stack maintains the current reduced state of the input string. Whenever closures, variables, or functions are recognized in the control stack, they are shifted to the object stack and are subsequently evaluated as follows:

- Evaluation of closures adds new variable bindings to the environment stack.
- Evaluation of variables uses bindings in the environment stack to change the variables to values.
- Evaluation of functions reduces them to a value or WHNF (§ 1.4.2).

Whenever a new environment is entered, the old environment and the contents of all other stacks are saved on the dump stack. Whenever an environment is exited, the old saved entries on the dump stack are restored to the respective stacks.

When complete, the environment, control, and dump stacks are empty, and the program result is located on the object stack.

Eager Interpreter SECD Stack operations to invoke an eager interpreter are shown in figure 4.

If the stacks are implemented as linked lists then $\mathsf{head}(stk)$ (where stk is a stack) is the same as (CAR stk) in LISP, $\mathsf{tail}(stk)$ is the same as (CDR stk), and $elt :: stk$ is equivalent to (CONS elt stk). If the stacks are implemented as arrays, then $\mathsf{head}(stk)$ is the element at the top of the stack, $\mathsf{tail}(stk)$ is stk after the top item has been removed, and $elt :: stk$ is stk after elt has been added to the top of stk. As is the case with the AST implementations (§ 2.1), the :: operator adds bindings to an environment, and the $||$ operator retrieves them (§ 2.1).

Figure 5 shows the SECD stack transitions used to reduce the expression $(\lambda y. + y\ y)\ 3$. The "Rule Applied" column refers to the stack transformation rules in figure 4.

Notice that transition 1.d in figure 5 adds the closure $[y, +y\ y,\ [nil]]$ to stack S (where y is a formal parameter of a lambda expression, $(+\ y\ y)$ is its body, and [nil] is the current environment. Next, transition 1.g uses the closure and the argument 3 (also on stack S) to add the binding $(y,\ 3)$ to stack E. Transition 1.g also saves the appropriate SECD stack contents on stack D. When transition 2 is applied at the bottom of figure 5, the contents of D is returned to the SECD stacks so

1. If stack C is not empty and $X = \mathsf{head}(\mathsf{C})$ then:

	Conditions	Next State
a	$X = k$ where k is a constant.	$(k :: \mathsf{S}, \mathsf{E}, \mathsf{tail}(\mathsf{C}), \mathsf{D})$
b	$X = b$ where b is a built-in function.	$(b :: \mathsf{S}, \mathsf{E}, \mathsf{tail}(\mathsf{C}), \mathsf{D})$
c	$X = v$ where v is a variable identifier and $\mathsf{E} \parallel v$ returns the binding of v in the environment E.	$((\mathsf{E} \parallel v) :: \mathsf{S}, \mathsf{E}, \mathsf{tail}(\mathsf{C}), \mathsf{D})$
d	$X = \lambda$-abstraction— $Param(x)$ is formal parameter, $body(X)$ is body.	$([param(X), body(X), \mathsf{E}] :: \mathsf{S}, \mathsf{E}, \mathsf{tail}(\mathsf{C}), \mathsf{D})$
e	$X = EXPR_1\ EXPR_2$ where $EXPR_1$ and $EXPR_2$ are expressions.	$(\mathsf{S}, \mathsf{E}, EXPR_2 :: (EXPR_1 :: (@ :: \mathsf{tail}(\mathsf{C}))), \mathsf{D})$
f	$X = @$ and the contents of stack S consists of a_2 (an argument), a_1 (another argument), and S' (remainder of stack S).	$((a_1, a_2) :: \mathsf{S}', \mathsf{E}, \mathsf{tail}(\mathsf{C}), \mathsf{D})$
g	$X = @$, and the contents of stack S consists of c (the closure $[v, B, \mathsf{E}']$), a (an argument), and S' (remainder of stack S).	$((),\ (v, a) :: \mathsf{E}', B, (\mathsf{S}', \mathsf{E}, \mathsf{tail}(\mathsf{C}), \mathsf{D}))$
h	$X = @$, and the contents of stack S consists of f (built-in function),$args$ (arg-list),and S' (remainder of stack S).	$(f(args) :: \mathsf{S}', \mathsf{E}, \mathsf{tail}(\mathsf{C}), \mathsf{D})$

2. If stack C is empty and $\mathsf{D} = (\mathsf{S}', \mathsf{E}', \mathsf{C}', \mathsf{D}')$, the next state is: $(\mathsf{head}(\mathsf{S}) :: \mathsf{S}', \mathsf{E}', \mathsf{C}', \mathsf{D}')$

Figure 4: State Transitions for an Eager SECD Interpreter

S	E	C	D	Rule Applied
()	[nil]	$(\lambda y. + y\ y)$ 3	()	1.e
()	[nil]	3, $\lambda y. + y\ y$, @	()	1.a
3	[nil]	$\lambda y. + y\ y$, @	()	1.d
[y, + y y, [nil]], 3	[nil]	@	()	1.g
()	[(y, 3)]	+ y y	(), [nil], (), ()	1.e
()	[(y, 3)]	y y, +, @	(), [nil], (), ()	1.e
()	[(y, 3)]	y, y, @, +, @	(), [nil], (), ()	1.c
3	[(y, 3)]	y, @, +, @	(), [nil], (), ()	1.c
3, 3	[(y, 3)]	@, +, @	(), [nil], (), ()	1.f
(3, 3)	[(y, 3)]	+, @	(), [nil], (), ()	1.b
+, (3, 3)	[(y, 3)]	@	(), [nil], (), ()	1.h
6	[(y, 3)]	()	(), [nil], (), ()	2
6	[nil]	()	()	

Figure 5: Eager Interpretation of $(\lambda y. + y\ y)$ 3 with SECD

execution can resume without further reference to the lambda expression.

A residual benefit of this saving and restoring of stacks is that only the current and saved environments are necessary to support SECD implementations (recall that AST implementations spawn sub-environments for each sub-expression (§ 2.1)). SECD simply saves the current environment on the dump stack when a lambda expression is entered, and then restores that environment when it is exited.

The two transition 1.c's in figure 5 access the environment for the binding of 3 to the variable y, each time placing the bound value on stack S. When complete, all of the lambda body arguments are available so transition 1.h can evaluate the function (+ 3 3). Since this is an eager interpreter, the arguments have already been reduced to WHNF (§ 1.4.2).

Lazy Interpreter The eager interpreter above can be converted to a partially lazy interpreter just by changing some of the transitions in figure 4.

First, transition 1.e is changed to:

$$(\mathsf{S},\ \mathsf{E},\ EXPR_1 :: (EXPR_2 :: (@ :: \mathsf{tail(C)})), \mathsf{D})$$

This simply reverses the positions of $EXPR_1$ and $EXPR_2$ from that of the eager interpreter. What this means is if $EXPR_1$ is a function and $EXPR_2$ is its argument, the eager interpreter will evaluate the argument first, and the lazy interpreter will evaluate the function first.

Aside from that, the only real difference is that functions and arguments (as well as closures and their arguments) of the lazy interpreter get pushed on stack S in reverse order from that of the eager interpreter. This is implemented by reversing the order they are handled as follows:

- in transition 1.f look for a_1 first followed by a_2,
- in transition 1.g look for the argument first followed by the closure, and
- in transition 1.h look for the argument-list first followed by the built-in function.

Eager evaluation is natural for SECD, and limited lazy evaluation is just as natural with the modifications to the driver function (transitions) described above. Unfortunately, full laziness with sharing, although possible, is just as complicated and cumbersome as it is in AST interpreters (§ 2.1).

SECD sharing is normally provided by adding off-stack structures that contain bindings of expressions to their reduced forms [Field and Harrison, 1988]. Each time an expression is encountered, the expression and its environment (e.g., (E, $[env]$)) are used to access the off-stack structure for a binding. If the expression was previously evaluated, a binding will be returned so the expression does not need to be evaluated. If no binding is returned, when the expression completes evaluation, its binding will be added to the off-stack structure.

This approach works, but at high cost. Each expression may contain sub-expressions, that in turn contain sub-expressions, and so forth. All of the expressions, and all of their sub-expressions, must be entered in this off-stack structure. Furthermore, since there are probably many environments, an expression evaluated in one environment is different from an expression evaluated in another environment (even if the binding is the same). Consequently, the space required to support full laziness in SECD is huge, and execution speed of fully lazy SECD systems is unacceptably slow.

2.3 Fixed-Combinators

Fixed-combinators are constructed from lambda expressions in a manner that removes occurrences of free variables. The transformations also remove lambda expressions so fixed-combinators are said to be *lambda free*. The benefit of lambda free expressions is they can be instantiated (§ 1.4.2) and then shared. This reduces copying and expression re-evaluation.

In contrast to the environment-based AST and SECD machines, fixed-combinators require no outside administrative overhead to maintain the environment, even if lazy evaluation is desired and they can be compiled [Peyton Jones, 1987; Field and Harrison, 1988].

Combinatory Logic (CL) was introduced by Schönfinkel [1924], and later was adapted to functional programming languages by Curry and Feys [1958]. The appeal of CL to functional programming languages is that a small fixed set of combinators can implement a fully lazy language. In theory, only two fixed-combinators, S and K are required, but the I (identity) and a wide assortment of other fixed-combinators can be included to optimize translation.

Lambda expressions can be transformed into S, K, and I combinators using the following transformations [Peyton Jones, 1987, Chapter 16]:

S	$(\lambda x.\ e_1\ e_2)$	$\Rightarrow$	$S\ (\lambda x.\ e_1)(\lambda x.\ e_2)$
K	$(\lambda x.\ c)$	$\Rightarrow$	$K\ c \quad (c \neq x)$
I	$(\lambda x.\ x)$	$\Rightarrow$	I

For example, the following is a transformation of $(\lambda x. + x\ x)\ 5$ (note that transformation rules are indicated above the transformation arrows):

$$
\begin{aligned}
& (\lambda y. + y\ y)\ 3 \\
\stackrel{S}{\Rightarrow}\ & S\ (\lambda y. + y)\ (\lambda y. y)\ 3 \\
\stackrel{S}{\Rightarrow}\ & S\ (S\ (\lambda y. +)\ (\lambda y.\ y))\ (\lambda y. y)\ 3 \\
\stackrel{I}{\Rightarrow}\ & S\ (S\ (\lambda y. +)\ I)\ (\lambda y.\ y)\ 3 \\
\stackrel{I}{\Rightarrow}\ & S\ (S\ (\lambda y. +)\ I)\ I\ 3 \\
\stackrel{K}{\Rightarrow}\ & S\ (S\ (K\ +)\ I)\ I\ 3
\end{aligned}
$$

The compiled expression $S\ (S\ (K\ +)\ I)\ I\ 3$ is in *Applicative Form* meaning it consists entirely of functions and arguments and it does not contain any lambda expressions. In fact, the compiled expressions are in *Constant Applicative Form* (CAF) since they do not contain variables. CAF is a desirable form because its application depends solely on its arguments, not on any free variables in its body. Thus translation either involves applying a CAF to its arguments, also in CAF, or applying a built-in function to its arguments, again in CAF.

Once compiled, CL expressions can be evaluated using the following reductions:

S	$S\ \mathsf{f}\ \mathsf{g}\ x$	$\rightarrow$	$\mathsf{f}\ x\ (\mathsf{g}\ x)$
K	$K\ c\ x$	$\rightarrow$	c
I	$I\ x$	$\rightarrow$	x

For example, the following is a reduction of $S\ (S\ (K\ +)\ I)\ I\ 5$ (note that the reduction type is indicated above each reduction arrow):

$$
\begin{array}{ll}
 & S\ (S\ (K\ +)\ I)\ I\ 3 \\
\xrightarrow{S} & S\ (K\ +)\ I\ 3\ (I\ 3) \\
\xrightarrow{S} & K\ +\ 3\ (I\ 3)\ (I\ 3) \\
\xrightarrow{K} & +\ (I\ 3)\ (I\ 3) \\
\xrightarrow{I} & +\ 3\ (I\ 3) \\
\xrightarrow{I} & +\ 3\ 3 \\
\xrightarrow{\delta} & 9
\end{array}
$$

Fixed-combinators can be directly implemented in hardware, bypassing a level of interpretation. Furthermore, it is not only possible to instantiate lambda bodies (avoiding copying and re-evaluation), but the instantiation is performed lazily. This means unused expressions are not evaluated. Therefore, CL is fully lazy. Finally, since only a few fixed-combinators must be recognized, a CL interpreter is simple to implement.

Unfortunately, however, CL compilation generates a large object program consisting of numerous fixed-combinators. Because the grain size of each fixed-combinator is small, a large number of steps is required to reduce a program. Furthermore, the reduction process creates many intermediate expressions that, following creation, are discarded almost immediately. Therefore, execution is still slow and consumes large amounts of transient storage. Caching schemes to speed up execution are hampered by the small task grain size.

2.4 Graph Reduction of Lambda Expressions

Another approach is to represent program expressions as a graph, and reduce the graph using reduction and conversion rules such as those presented for lambda expressions (§ 1.4.3). This is called *graph reduction* [Peyton Jones, 1987; Field and Harrison, 1988]. Although graph reduction can be applied to programs with other intermediate forms (§ 2.6), this subsection only considers the graph reduction of lambda expressions.

It has already been shown that lambda expressions can be represented as binary trees called abstract syntax trees, and that these trees, in turn, can be reduced by AST interpreters (§ 2.1). If this type of binary tree structure is modified to allow the sharing of common sub-expressions, the tree is transformed into a graph. Such graphs have the advantage that common sub-expressions need be evaluated only once. Therefore, suspensions (§ 2.1) or supporting stacks (§ 2.2) are not needed to provide sharing during lazy evaluation.

Graph Organization Graph nodes fall into one of the following three categories:

- *data*—variables, values or lists of variables and values.
- *functions*—built-in functions or lambda abstractions.
- *application* (@)—connection nodes (i.e., applies the argument at right-child to the function at left-child).

A special node called the *Root* is either a data, function, or @-node located at the base (root) of a sub-expression's graph, a position where evaluation begins. This evaluation, called *sparking* (because it sets activities in motion like setting a match to the root), proceeds as follows. While the root node is @, a search is made downwards from the root along the leftmost path (always selecting the

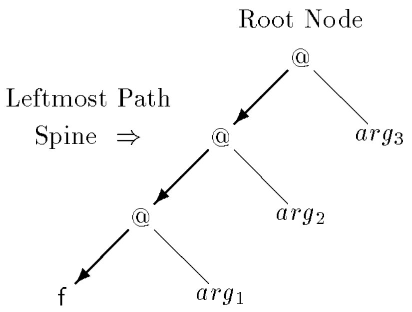


Figure 6: Graph Spine

leftmost child), bypassing @-nodes, until a function is located. This is called a *spine search* because the leftmost path looks like the spine of a rib cage (where diverging edges to rightmost nodes along the way represent the ribs). For example, the spine in figure 6 is the path from the root to node f and the ribs are the edges connecting the spine with arg_1, arg_2, and arg_3. A spine search of the graph in figure 6 locates the function f. Once located, the function becomes a *reduction expression* (redex). Each redex in the graph is an independent and non-interfering component of work.

Beta-Reduction If the spine search locates a λ node, a β-reduction is performed as follows:

1. If eager evaluation is used (§ 1.3), the lambda argument must be sparked and reduced before proceeding. This is accomplished by conducting a spine search from the argument's root node and then applying appropriate reductions described here and in succeeding paragraphs to reduce the argument to WHNF (§ 1.4.2). If lazy evaluation is used, the argument should *not* be sparked. Instead, go immediately to step 2.

2. Copy the lambda expression's body into a new data area.

3. Substitute the actual argument for the formal parameter wherever it appears in the copied body of the lambda expression.

4. Overwrite the old redex root with the root of the copied body.

For example, the β-reduction

$$(\lambda y. + y\ y)\ 3 \quad \xrightarrow{\beta} \quad + 3\ 3$$

is performed graphically as shown in figure 7 (a).

Notice that y is shared in the lambda body. As a consequence, the argument 3 is applied to one location instead of two during β-reduction.

Since β-reduction overwrites the root node of the old graph with the root node of the new graph, when β-reduction is complete, any nodes of the old graph that do not have other connections to the overall program graph, will never be used again during program evaluation. These nodes are garbage (i.e., they consume space unnecessarily and should be reclaimed).

Nodes in the old graph are connected if they are shared with components in the overall program graph. For example, if the old graph is $(\lambda y.\ b)\ a)$, and an external graph component accesses b_c (where b_c is a component of to old lambda body b), then the nodes in b_c *are* connected to the larger program graph and *are not* garbage. Assuming b_c is the only shared component, other nodes in $(\lambda y.\ b)\ a$ are garbage.

Delta-Reduction If the spine search locates a built-in function (such as $+$, $*$, or if), a δ-reduction is performed as follows:

1. A table is consulted to determine the number and types of arguments required by the built-in function.

2. Arguments are found at the right-child of @ nodes encountered when retracing the redex spine from the built-in function to the root node.

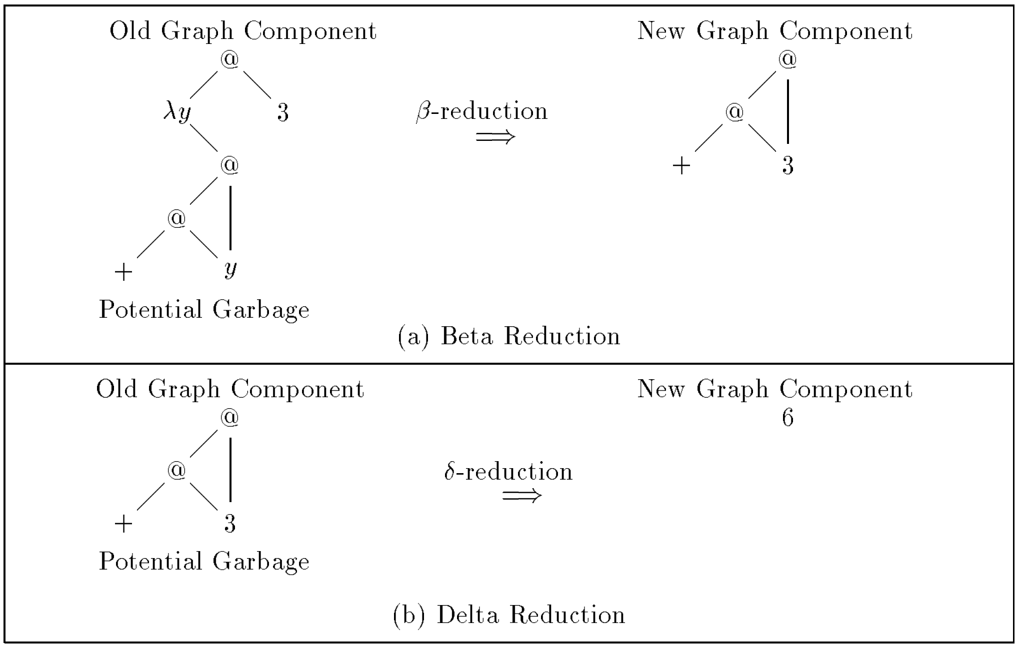


Figure 7: Graphical Representation of Beta- and Delta-reductions

3. If lazy evaluation is used, unevaluated arguments must be sparked and reduced to WHNF before proceeding. If eager evaluation is used, they are already in WHNF.
4. Evaluate the function using the appropriate arguments.
5. Overwrite the root node of the built-in function redex with the a node containing the evaluation results.

For example, the delta-reduction

$$+\ 3\ 3 \quad \xrightarrow{\delta} \quad 6$$

is performed graphically as shown in figure 7 (b).

If the argument 3 was an expression instead of a value, sharing would cause it to be evaluated once instead of twice during δ-reduction.

The result, 6, overwrites the root node of the old graph. Therefore, any components of the old graph that are not shared with the overall program graph are garbage.

Organization Implications Lazy evaluation with normal order reduction (§ 1.4.2) is natural in graph reduction systems. When a spine search locates a lambda function, it is more natural to complete the β-reduction with an unevaluated argument rather than reducing the argument first. However, both eager and lazy evaluation are possible. In fact, it is possible to mark graph function and application nodes as either strict or non-strict during intermediate translation, and invoke the specified evaluation method during execution. This leads to mixed-order evaluation which is able to draw upon the advantages of both techniques.

Parallel evaluation is also quite natural in graph systems. As was noted earlier, reduction expressions are independent and non-interfering components of work. As redexes are identified, the graph component corresponding to it can be transferred to another processor where its root node is sparked. In this way, multiple processors can be evaluating a program concurrently.

For example, during the eager evaluation of a lambda abstraction, the β-reduction and evaluation of the argument can be performed in parallel. Additionally, evaluation of the argument could recursively spark more β-reductions and argument evaluations theoretically leading to an unlimited number of parallel activities. Unfortunately, eager evaluation enjoys no parallelism during δ-reduction. All of the arguments are evaluated, so δ-reduction is the only remaining activity.

The situation is virtually reversed for lazy evaluation. Lazy evaluation never sparks arguments until δ-reduction, so no parallelism is possible during β-reduction, but the arguments can be sparked in parallel during δ-reduction. This is extremely limited parallelism in comparison to eager evaluation. One reason is that δ-reduction takes place later than β-reduction, so lazy-evaluation's parallelism is concentrated at the end of the reduction process. Also, the parallelism is concentrated within only the expression being δ-reduced. It doesn't fan out to other graph expressions like eager evaluation's parallelism does.

Parallelism will be covered more thoroughly later (§ 4).

Garbage Collection It is important to recognize that large amounts of garbage are normally generated during β- and δ-reductions. The garbage problem implies using mechanisms such as a *garbage collector* to detect and reclaim unused space. Unfortunately, collection mechanisms add overhead that diminishes system performance. Garbage collection techniques are examined more thoroughly in the parallel section (§ 4.7).

Compilation Compilers are used to translate languages from a form that is convenient for humans to understand and manipulate (such as Haskell) to a form that is easier for machines to execute. One alternative is the compilation of high-level languages to lambda expressions. However, if lambda expressions contain free variables, they cannot be evaluated until the free variable bindings have been determined (§ 1.4.2). Frequently these bindings are not applied until run-time, and late bindings make compilation impossible. If compilation is still desired, the bindings must be made available earlier by using mechanisms such as closures (§ 2.1 and § 2.2). These closures store the bindings in an environment as they come available. A compiler can access bindings from the environment as they are needed. Unfortunately, the implementation of closures involves administration that leads to poor performance, so this form of compilation may not be a realistic alternative.

Unnecessary Copying By sharing graph nodes, the recomputation of common sub-expressions can be avoided, but graph implementations still can encounter a loss of laziness from unnecessary copying. For example, consider the β-reduction of $(\lambda y.\ E_1)\ E_2$ shown in figure 8 (a). Also, assume that there are no occurrences of the formal parameter y in E_1. In that case, E_1 would be copied as is from its current position to the root node as shown. This copying is unnecessary and extremely undesirable if E_1 is a large structure. If E_1 is shared, there are two accessible copies of E_1 following β-reduction (see right-hand side of figure 8 (a)).

A solution is to use an indirection node as shown in figure 8 (b). Instead of overwriting the old root with a copy of E_1, the indirection node Θ is placed there instead. Θ points to the original E_1 obviating the need to copy it. Although indirection nodes do solve the copying problem they exhibit the following problems:

- they require some administration to detect circumstances where they can be applied,

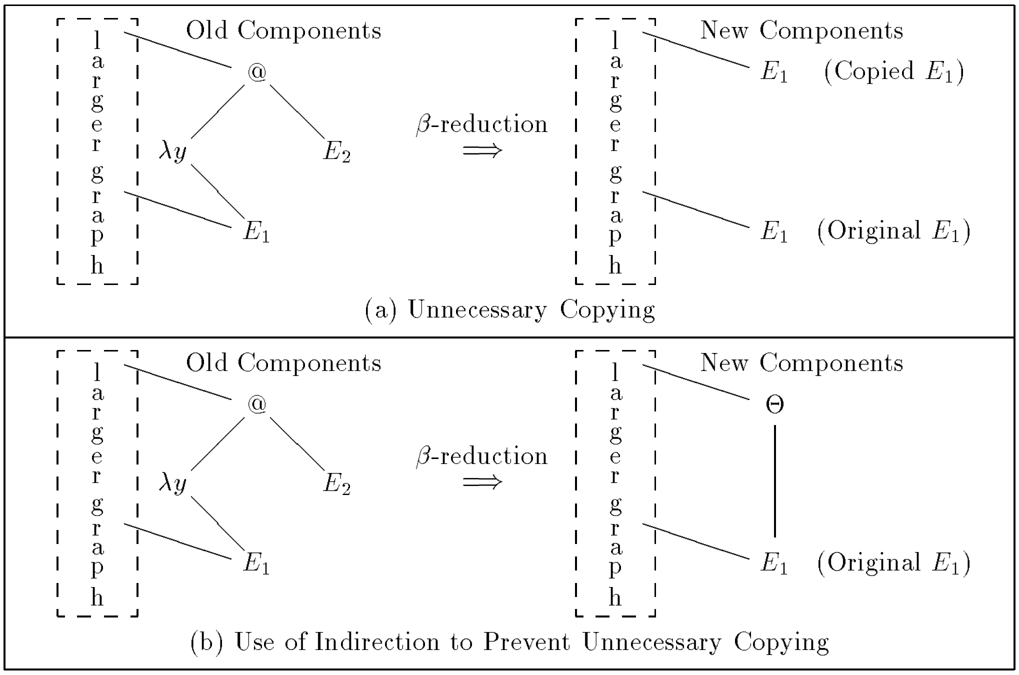


Figure 8: Indirection

- they can lead to long chains of indirection if it is necessary to use them frequently, and
- there is a loss of efficiency when components are accessed indirectly.

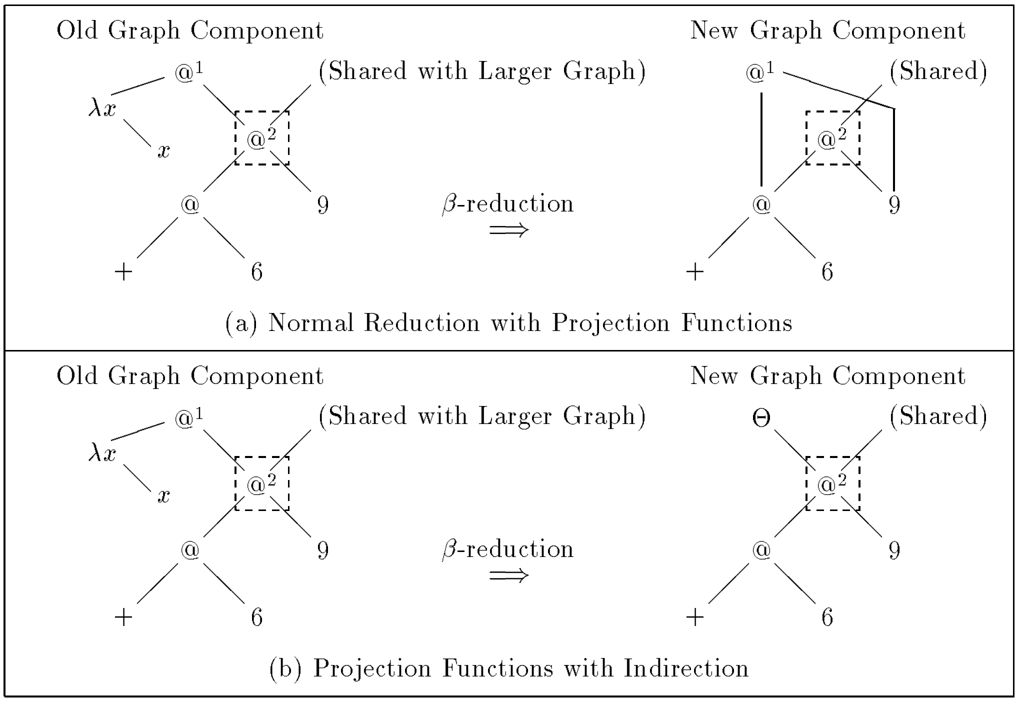
A greater problem arises when there *are* occurrences of y in E_1. Then, β-reduction changes E_1. If the change is made in place and an indirection node points to it as shown in figure 8 (b), the changed copy will be the one accessed by other components sharing E_1, although they are expecting to access the original copy. In this case, indirection nodes should not be used. Instead, the modified E_1 should be copied to the root (e.g., figure 8 (a)).

Projection Functions Projection functions return unmodified components of their arguments. Those arguments can be reevaluated unnecessarily if they are shared. For example, consider the expression:

$$(\lambda x.\ x)\ (+\ 6\ 9) \tag{2}$$

and assume the argument (+ 6 9) is shared with another component of the graph. A normal β-reduction is shown in figure 9 (a). Notice that when the reduction is complete, node $@^1$ (the root node where expression 2 above will evaluate (+ 6 9)), is different from node $@^2$ (the root node where the sharing component will evaluate (+ 6 9)). Consequently, the result computed by one is not available to the other. Therefore, recomputation is necessary.

As was the case with the unnecessary copying problem above, one solution is to employ indirection nodes. For example, The indirection node Θ in figure 9 (b) is expression (2)'s root node. With this change, it makes no difference whether (+ 6 9) is evaluated indirectly through Θ, or directly by the sharing expression. Once one completes the computation, recomputation is unnecessary. The



(a) Normal Reduction with Projection Functions

(b) Projection Functions with Indirection

Figure 9: Projection Functions

problems encountered when using indirection nodes have already been identified.

2.5 Super-Combinators

A lambda expression that contains no free variables can become a super-combinator [Hughes, 1984]. Unlike the fixed-combinators discussed earlier (§ 2.3), super-combinators are formed from the user's program, rather than from a fixed set of combinators. The lambda expression is normally transformed to a special super-combinator format by assigning it a name, associating the bound variables with it, and setting it equal to the body of the lambda expression. For example, since all variables are bound in the lambda expression to the left of the arrow below, it can be transformed to super-combinator notation shown at the right of the arrow.

$$\lambda x.\ \lambda y. + y\ x \quad \Longrightarrow \quad \$\mathsf{Y}\ x\ y\ = +y\ x$$

Like Combinatory Logic (CL), (§ 2.3), super-combinators support fully lazy evaluation and compilation without outside administrative support such as closures and suspensions (§ 2.1 and § 2.2). However, the task grain size of super-combinators is larger than CL, so its performance can be better.

As is the case with CL, super-combinators involve the translation of lambda expressions to Constant Applicative Form (CAF), a form that contains no free variables and no λ's (see § 2.3, [Peyton Jones, 1987; Field and Harrison, 1988]. However, unlike CL, super-combinators do not have fixed types (e.g., S, K , I). Instead, an unlimited number of super-combinator types are generated dynamically as a program's lambda expressions are translated.

Since super-combinators contain no free variables, they are not dependent on outside influences. Therefore, they can be instantiated (§ 1.4.2) and shared.

Lambda Lifting Prior to Translation
Free variables are removed from lambda expressions using a process called lambda lifting (§ 1.4.4). Therefore, all lambda expressions can be converted to super-combinators.

For example, consider the conversion of:

$$(\lambda x.\ (\lambda y.\ +\ y\ x)\ x)\ 4$$

The sub-expression:

$$(\lambda y. + y\ x)$$

is not a super-combinator because it contains the free-variable x. However, lambda lifting removes the free-variable as follows:

$$\begin{array}{ll} & (\lambda y. + y\ x) \\ \xrightarrow{\beta_a} & (\lambda x.\ \lambda y. + y\ x)\ x \\ \xrightarrow{\alpha} & (\lambda w.\ \lambda y. + y\ w)\ x \end{array}$$

where β_a is a beta-abstraction to remove the free-variable and α is an α-conversion (§ 1.4.3) to change the name of x inside the parentheses (where it is bound) to w so it is not confused with the x outside the parentheses (which is free). Substituting this result in the original expression yields:

$$(\lambda x.\ (\lambda w.\ (\lambda y. + y\ w))\ x\ x)\ 4$$

which can be transformed to super-combinators as follows.

First, note that all variables are bound in:

$$(\lambda w.\ (\lambda y. +\ y\ w))$$

so it can be rewritten as the super-combinator:

$$\mathsf{\$Y}\ w\ y\ = +y\ w$$

Then, the original expression is transformed to:

$$\begin{array}{l} \mathsf{\$Y}\ w\ y\ = +y\ w \\ (\lambda x.\ \mathsf{\$Y}\ x\ x)\ 4 \end{array}$$

Next, notice that $(\lambda x.\ \mathsf{\$Y}\ x\ x)$ is also a super-combinator. Assign it the name $X. The expression then is:

$$\begin{array}{l} \mathsf{\$Y}\ w\ y\ = +y\ w \\ \mathsf{\$X}\ x\ =\ \mathsf{\$Y}\ x\ x \\ \mathsf{\$X}\ 4 \end{array}$$

Finally, since $X 4 is another super-combinator, assign it the name $PROG. The final translated code is comprised entirely of super-combinators:

$$\begin{array}{l} \mathsf{\$Y}\ w\ y\ = +y\ w \\ \mathsf{\$X}\ x\ =\ \mathsf{\$Y}\ x\ x \\ \mathsf{\$PROG}\ =\ \mathsf{\$X}\ 4 \\ \mathsf{\$PROG} \end{array}$$

Execution of the translated program is shown below (note that labels on execution step arrows identify the reduction actions accomplished at each step and subscripts identify variables involved in the reduction).

$$\begin{array}{ll} & \mathsf{\$PROG} \\ \rightarrow & \mathsf{\$X}\ 4 \\ \xrightarrow{\beta_x} & \mathsf{\$Y}\ 4\ 4 \\ \xrightarrow{\beta_{y,w}} & +\ 4\ 4 \\ \xrightarrow{\delta} & 8 \end{array}$$

Lambda Lifting During Translation
Instead of lambda lifting prior to conversion to super-combinators, the lambda lifting process can be performed concurrently with translation. The following algorithm performs both the lifting and translation.

REPEAT (until there are no more lambda expressions)

1. Choose a lambda expression that contains no other lambda expressions.
2. Assign an arbitrary super-combinator name to the selected lambda expression.
3. Assign all bound and free variables found in the lambda expression body as the super-combinator's parameters.

Using this algorithm, the expression:

$$(\lambda x.\ (\lambda y. + y\ x))$$

would be compiled as follows. The innermost lambda expression is $(\lambda y. + y\ x)$, and when translated, the original expression becomes:

$$\begin{array}{l} \mathsf{\$Y}\ x\ y\ =\ +\ y\ x \\ \lambda x.\ \mathsf{\$Y}\ x \end{array}$$

Then, when $(\lambda x.\ \$\mathsf{Y}\ x)$ is translated the expression becomes:

$$\begin{aligned}&\$\mathsf{Y}\ x\ y\ =\ +\ y\ x\\&\$\mathsf{X}\ x\ =\ \$\mathsf{Y}\ x\\&\$\mathsf{X}\end{aligned}$$

where the final entry, $X, invokes the code.

However, eta conversion (§ 1.4.3):

$$\$\mathsf{X}\ x \quad \overset{\eta}{\leftrightarrow} \quad \$\mathsf{Y}\ x$$

implies that the super-combinators $X and $Y are equivalent, so the translated code can be reduced to:

$$\begin{aligned}&\$\mathsf{Y}\ x\ y\ =\ +\ y\ x\\&\$\mathsf{Y}\end{aligned}$$

Maximally Free Expressions Both of the translation methods described above involve lambda lifting. The first performs the lifting before translation, and the second performs it during translation. In both cases, only the free variables are lifted from the expression. This subsection will show that better sharing is possible when lambda lifting removes the entire sub-expressions containing free variables rather than just the free variables.

Sub-expressions that do not contain bound variables are said to be *free* expressions. If a free expression is not contained in any other free expression it is said to be *maximally free*. For example, the underlined sub-expressions below are maximally free in λx:

- $(\lambda x.\ \underline{\mathsf{sqrt}\ x})$
- $(\lambda x.\ \underline{\mathsf{log}\ 20})$
- $(\lambda y.\ \lambda x.\ \underline{(*\ y\ y)})$ (maximally free in λ x, not λ y)

Now consider the lambda expression:

$$(\lambda f. + (f\ 4)(f\ 7))(\lambda x.\lambda y. + y(\mathsf{sqrt}\ x))9) \quad (3)$$

which translates to the following super-combinators when lambda lifting only removes free variables:

$$\begin{aligned}&\$\mathsf{T}\ x\ y\ =\ +\ y\ (\mathsf{sqrt}\ x)\\&\$\mathsf{F}\ =\ \$\mathsf{T}\ 9\\&\$\mathsf{PROG}\ = +(\$\mathsf{F}\ 4)\ (\$\mathsf{F}\ 7)\\&\$\mathsf{PROG}\end{aligned}$$

Notice that the expression sqrt 9 is evaluated twice during execution of that program:

$$\begin{aligned}&\quad \$\mathsf{PROG}\\\longrightarrow&\quad +\ (\$\mathsf{F}\ 4)\ (\$\mathsf{F}\ 7)\\\longrightarrow&\quad +\ (\$\mathsf{T}\ 9\ 4)\ (\$\mathsf{T}\ 9\ 7)\\\longrightarrow&\quad +\ (\$\mathsf{T}\ 9\ 4))\ (+\ 7\ (\mathsf{sqrt}\ 9))\\\longrightarrow&\quad +\ (\$\mathsf{T}\ 9\ 4))\ (+\ 7\ 3)\\\longrightarrow&\quad +\ (\$\mathsf{T}\ 9\ 4))\ 10\\\longrightarrow&\quad +\ (+\ 4\ (\mathsf{sqrt}\ 9))\ 10\\\longrightarrow&\quad +\ (+\ 4\ 3)\ 10\\\longrightarrow&\quad +\ 7\ 10\\\longrightarrow&\quad 17\end{aligned}$$

If expression (3) above is recompiled using lambda lifting to remove maximally free expressions, evaluation of the sub-expression sqrt 9 will be shared. Only the following small adjustments to step 3 of the algorithm on page 23 are required:

3. Choose a name for each maximally free expression and then assign both bound variables and maximally free expressions as the super-combinator's parameters.

For example, consider the sub-expression:

$$\lambda x.\ \lambda y. + y\ (\mathsf{sqrt}\ x)$$

When the innermost abstraction (λy) is processed, note that the variable y in the lambda body is bound and $\mathsf{sqrt}\ x$ is a maximally free expression. The super-combinator for λy must be given a name, say $T1. Its maximally free expression must also be given a name, say rootx. Then λy becomes the super-combinator:

$$\$\mathsf{T1}\ \mathsf{rootx}\ y\ =\ +\ y\ \mathsf{rootx}$$

and λx becomes

$$\lambda x.\ \$\mathsf{T1}\ (\mathsf{sqrt}\ x)$$

If the λx super-combinator is given the name $\mathsf{\$T2}$, it becomes:

$$\mathsf{\$T2}\ x \ = \ \mathsf{\$T1}\ (\mathsf{sqrt}\ x)$$

The entire super-combinator program for expression (3) is:

$$\begin{array}{l}
\mathsf{\$T1\ rootx}\ y \ = \ +\ y\ \mathsf{rootx} \\
\mathsf{\$T2}\ x \ = \ \mathsf{\$T1}\ (\mathsf{sqrt}\ x) \\
\mathsf{\$F} \ = \ \mathsf{\$T2}\ 9 \\
\mathsf{\$PROG} \ = +(\mathsf{\$F}\ 4)\ (\mathsf{\$F}\ 7) \\
\mathsf{\$PROG}
\end{array}$$

Instead of instantiating the maximally free expression, a pointer to the single shared instance is substituted in its place. This idea was first proposed by Wadsworth in 1971 [Wadsworth, 1971] and was adapted to super-combinators by Hughes in 1984 [Hughes, 1984].

It is important to recognize that there is sharing during execution. Shared expressions are underlined in the execution sequence below so they stand out. Because of the sharing, (sqrt 9) is only evaluated once.

$$\begin{array}{ll}
 & \mathsf{\$PROG} \\
\longrightarrow & +(\mathsf{\$F}\ 4)\ (\mathsf{\$F}\ 7) \\
\longrightarrow & +(\underline{\mathsf{\$T2}\ 9}\ 4)\ (\underline{\mathsf{\$T2}\ 9}\ 7) \\
\longrightarrow & +(\overline{\mathsf{\$T1}\ \underline{(\mathsf{sqrt}\ 9)}}\ 4)\ (\overline{\mathsf{\$T1}\ \underline{(\mathsf{sqrt}\ 9)}}\ 7) \\
\longrightarrow & +(\mathsf{\$T1}\ \overline{\underline{(\mathsf{sqrt}\ 9)}}\ 4)\ (+\ 7\ \overline{\underline{(\mathsf{sqrt}\ 9)}}) \\
\longrightarrow & +(\mathsf{\$T1}\ \underline{3}\ 4)\ (+\ 7\ \underline{3}) \\
\longrightarrow & +(\mathsf{\$T1}\ 3\ 4)\ 10 \\
\longrightarrow & +(+\ 4\ 3)\ 10 \\
\longrightarrow & +7\ 10 \\
\longrightarrow & 17
\end{array}$$

2.6 Compilation

Compilation performs a large share of the analysis required to transform high-level language programs to machine-executable form. Consequently, compiled code is easy to interpret and tends to execute quickly. This subsection discusses imperative style, source-to-source, and super-combinator compilers.

Imperative Style Compilers Functional languages can be compiled in the same manner as imperative languages, but they usually are not. Two reasons for this are the functional paradigm's use of higher-order functions and the occurrence of free variables in expressions during lazy evaluation.

Higher-order functions (§ 1.3) allow functions themselves to be passed as parameters. In this way, functions can be used to construct more powerful functions.

Some imperative languages also pass functions (and procedures) as parameters, referring to them as *formal procedures* [Fisher and Leblanc, 1988; Aho *et al.*, 1986; Sebesta, 1989]. Unfortunately, formal procedures add considerable overhead to the compile and run-time organization of imperative languages. Since functional languages rely very heavily on function parameters, they can ill afford to contend with these cumbersome implementation procedures.

Also, functional-language free variables (§ 1.4.2) are similar to global variables in imperative languages. Imperative languages allocate space for both global and local variables in run-time data areas called *activation records*. Compile time addresses that access these data areas consist of a lexical level designator (to access the appropriate activation record) and an offset (to identify the variable's position in the activation record). If all variables were passed call-by-value (§ 1.4.2), functional languages could use the activation record scheme too, accessing the required free-variable value bindings from appropriate activation records. However, functional languages use call-by-name, and it is call-by-name that causes the biggest problem. In call-by-name, bindings are to expressions not values. These expressions cannot be stored in an activation record because they are in the form of code, not a single value, and the identity of the expression that will be bound is unknown at compile time. The few imperative languages that use call-by-name store code for the expression (called *thunks*) in static memory, and invoke the code when

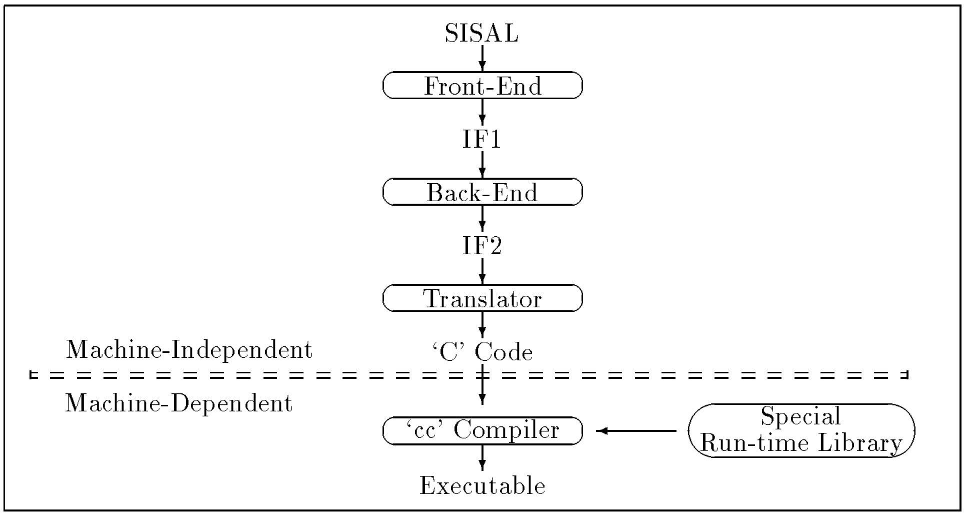


Figure 10: SISAL Translation Organization

the parameter is accessed. Thunks are slow and inefficient. They are not an acceptable alternative for languages that frequently use call-by-name.

Note, however, that a sophisticated functional compiler *could* identify sections of code using only call-by-value parameter passing and compile that code imperatively.

Source-to-Source Translators An approach used by *Streams and Iterations in a Single Assignment Language* (SISAL) [Haines and Böhm, 1991] and some versions of Haskell [Hudak and Wadler, 1988] is to translate the functional source code (e.g., SISAL) to source code in an imperative language (e.g., 'C') and then compile the imperative source to machine executable form. This method has the advantage that if the target source is a popular language like 'C', the resulting implementation is highly portable.

SISAL uses 'C' as its target language. Its translation organization shown in figure 10 is comprised of a front-end, back-end, and run-time system [Haines and Böhm, 1991]. The front-end ensures syntactic correctness, translates SISAL programs to an intermediate form called *IF1* [S.K.Skedzielewski and Glauert, 1985]. The back-end outputs another intermediate form called *IF2* that includes storage requirement directives and code optimizations (such as copy removal). IF2 is in turn is translated to 'C' code which is compiled in the environment of a specialized run-time library. Among other things, the run-time library generates memory management calls to the host operating system which satisfy IF2's storage directives. This provides the added benefit of a simple yet powerful dynamic storage allocation scheme.

The question of course is whether source-to-source translation sacrifices performance as a result of the translation. Timings reported for SISAL in [Böhm *et al.*, 1991] indicate that it does not. In fact, SISAL's execution speed is quite good.

Super-Combinator Compilers An alternative to imperative-style compilation is to remove free variables using lambda lifting and transform lambda expressions to super-combinators. *FPM* [Field and Harrison, 1988, Chapter 15] and the *G-Machine* [Augustsson, 1984; Johnsson, 1984] use this approach to compilation.

FPM was originally developed to support the original Edinburgh version of Hope [Burstall *et al.*, 1980]. It implements purely eager evaluation and can be viewed as an optimized version of the SECD machine (§ 2.2). Hope code is lambda lifted and translated to an intermediate form called FC [Baily, 1985] and then is further translated to compiled code.

The limitation to eager evaluation is a serious problem for FPM, one that undermines its usefulness. Since the G-Machine can allow both eager and lazy evaluation, it is used as the implementation example for super-combinator compilers.

The *G-machine* was developed at Chalmers Institute of Technology, Göteborg, Sweden by Augustsson and Johnsson. It uses lazy evaluation to execute compiled code, called *G-code*, derived from super-combinators. The machine can be adapted to execute some expressions eagerly. Compilation is straightforward since there are no free variables to contend with. The compiled instructions use a stack to successively instantiate (§ 1.4.2) bodies of a super-combinator program. As a consequence of the instantiation, the program is reduced to normal form [Field and Harrison, 1988; Peyton Jones, 1987; Peyton Jones and Lester, 1992].

The following is a list of G-code instructions along with explanations of their operation.

- `add`—Pops the top two items from the stack, adds them, and then pushes the result on the stack in their place.
- `mkap`—Pops the top two items from the stack and pushes an application node in their place, with the first popped item as the left-child (function) node and the second as the right-child (argument) node.
- `push` i—Pushes the expression at stack position i on the stack (*stacktop* is 0, $stacktop - 1$ is 1, ..., $stacktop - i$ is i).
- `pushglobal` j—Pushes the global expression j on the stack.
- `unwind` 1—Performs a spine search (§ 2.4) starting at *stacktop*. Pushes the function found on the stack.
- `unwind` 2—Consult a table to determine the number of arguments required by the function found at the previous instruction (`unwind` 1). Locate those arguments at the right child of application nodes proceeding up the spine from the function towards the root. Pop the function off the stack and push the arguments on in its place.
- `update` k—Place *stacktop* in temporary t. Pop $k+1$ items from the stack. Push t on the stack in their place.

A lazy super-combinator compilation algorithm is is shown in figure 11:

To demonstrate the G-machine's operation, consider the lambda expression:

$$(\lambda y. + y\ y)\ 3$$

which can be translated to the following super-combinators (§ 2.5):

$$\begin{aligned} \$\mathsf{D}\ y &= +\ y\ y \\ \$\mathsf{P} &= \$\mathsf{D}\ 3 \\ \$\mathsf{P} & \end{aligned}$$

which in turn is compiled to the G-Code instructions shown in figure 12.

Execution of those instructions is shown in figure 13. This example clearly demonstrates how instantiation contributes to the reduction process. Notice that the first two instructions, `pushglobal` 3 and `pushglobal` $D, push the body of super-combinator $P on the stack. Then, `makp` makes an application node out of it and `update` 1 replaces $P with the application node. As a result of these instructions, $P's body is instantiated and *stacktop* is updated so it points to that application node (which is the root of that instantiated body).

1. Start with a super-combinator at the *stacktop*.
2. Use pushglobal to push super-combinator body on stack.
3. Recursively apply mkap to top two items in stack until all items in the super-combinator body are combined into a subgraph with *stacktop* pointing to the root application node of the subgraph.
4. Use update to replace the super-combinator in 1. above with the root application node of the super-combinator body.

REPEAT

a. Conduct spine search using unwind 1 to locate a function.

b. IF the function locate is a super-combinator THEN

(1) Use unwind 2 to place super-combinator arguments in order at *stacktop*.

(2) Use push or pushglobal to push super-combinator body on stack. Push is used when item is already on stack; pushglobal is used when it is not on the stack.

(3) Apply step 3. above.

(4) Use update to replace the super-combinator in b. above with the root application node of the super-combinator body.

c. ELSE IF a built-in function is located THEN

(1) Use unwind 2 to place the built-in function arguments in order at the *stacktop*.

(2) Apply built-in function to arguments and place result at *stacktop*.

(3) Use update to replace built-in function in c. above with the result.

UNTIL (The expression is evaluated—result is at *stacktop*)

Figure 11: Super-combinator Compilation Algorithm

pushglobal 3	Push 3 on stack	
pushglobal $D	Push $D on stack	Instantiate body of $P
mkap	Make application node out of $D 3	
update 1	Replace $P with its instantiated body	
unwind 1	Conduct spine search to locate $D	
unwind 2	Locate $D's argument	
push 0	Push first argument in $D's body on stack	
push 1	Push second argument in $D's body on stack	Instantiate body of $D
pushglobal +	Push + on stack	
mkap	Make application node out of + 3	
mkap	Make application node out of + 3 3	
update 2	Replace root node with $D's instantiated body	
unwind 1	Conduct spine search to locate +	
unwind 2	Locate +'s argument	Perform addition
add	Perform addition	
update 2	Replace root node with result	

Figure 12: G-Code for $(\lambda y. + y\ y)\ 3$

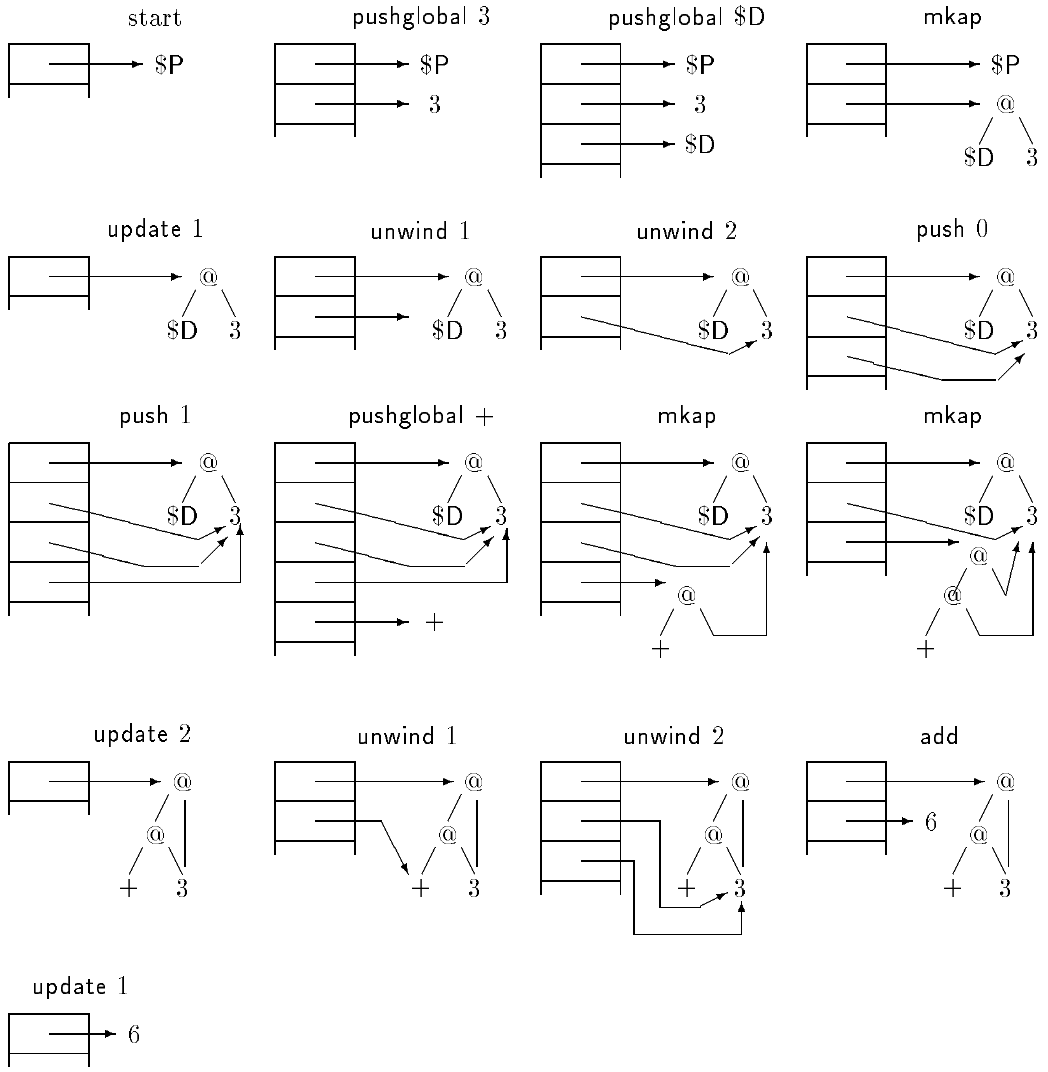


Figure 13: Stack Execution of Compiled Super-Combinator Code

The next instruction, unwind 1, performs a spine search that locates the super-combinator \$D. This determines that \$D's body, $+ \ y \ y$, will be the next expression instantiated on the stack. However, before beginning the instantiation, unwind 2 locates \$D's argument 3. The next three instructions, push 0, push 1, and pushglobal + add the body elements to the stack, but notice that instead of pushing the variable y on the stack, push 0 and push 1 push the argument identified by unwind 2 instead. The two successive mkap instructions complete the instantiation, and update 2 places the root of the instantiated body at the *stacktop*.

The cumulative effect of instantiating \$D 3 and $+ \ y \ y$ is the completion of a β-reduction on the original lambda expression. That is, the lambda body, $+ \ y \ y$, is positioned at the top of the stack with the argument 3 applied to y. Also, notice that the argument 3 is shared. Therefore, if it requires evaluation, the reduction will only be performed once. The remaining instructions unwind 1, unwind 2, add, and update 1 perform a δ-reduction on $+ \ 3 \ 3$ and place the result, 6, on top of the stack.

Combinator compilation schemes represent an organized approach to compilation. Implementations like the G-machine require little overhead in comparison with environment-based systems such as AST interpreters (§ 2.1) or SECD (§ 2.2). The stack used with the G-machine neatly organizes operations, and provides the additional benefit of allocating and de-allocating memory.

On the negative side, although the task grain size is larger than CL (§ 2.3), it is still small enough to be the subject of criticism. Also, all of the copying and sharing problems that apply to lambda expression graph reduction also apply to super-combinator graph reduction (§ 2.4).

3 Optimizations

All of the implementation techniques presented so far have limitations that diminish their usefulness. Some of these limitations are listed below.

- "Pure" functional languages do not use iteration to implement repetition. Since other repetition methods such as recursion are slower and tend to use more space than iteration, functional languages usually are slower and consume more space than imperative languages during repetition. Subsection 3.1 below demonstrates that a technique called tail recursion elimination can convert some recursive functions to either iterative or non-tail recursive functions.

- Although "full" laziness significantly reduces copying and the re-evaluation of expressions, it does not provide for the sharing of equivalent function calls in separate parts of a program or equivalent function calls generated during recursion. Functions such as the Fibonacci function that do not take advantage of this sharing can have a much higher order of complexity than those that do share or their iterative equivalents. Subsection 3.2 describes how memoization changes the form of a function so it acts like a memo pad remembering and reusing previously computed results.

- Multiple instances of the same variable in a lambda body are usually not shared if the expression to which the variable is bound contains free variables (§ 1.4.2). The problem is those free variables will receive different bindings during the reduction process. Subsection 3.3 demonstrates that the free variables can be identified and handled separately during the reduction, allowing other variables in the expression to be shared.

- Reduction steps that could be performed at compile time are delayed until execution time. Subsection 3.4 demonstrates that some function definitions can be partially evaluated with respect to one or more parameters during compilation, saving work at execution time.
- The implementations generally seek to implement either fully lazy or fully eager evaluation, whereas a mixed-order reduction scheme might yield more optimal performance. Subsection 3.5 analyzes several mixed-order evaluation techniques.

3.1 Tail Recursion Elimination

The last thing a *tail recursive* function does is call itself [Appleby, 1991; Davie, 1992; Field and Harrison, 1988; Peyton Jones, 1987]. Computation of a tail recursive function's value cannot begin until the final recursive call is completed. This delays the computation unnecessarily, wasting time and consuming space. Consider the following example (adapted from [Peyton Jones, 1987]) that sums a list of numbers. Note that `n` is a number, `ns` is a list of numbers, and square brackets (i.e., [and]) enclose lists.

```
sum  [ ]    = 0
  '  (n:ns) = n + (sum ns)
```

Notice that the recursive call to `sum` occurs at the end of each reduction step in the following evaluation of `sum [1,2,3]`:

```
    sum [1,2,3]
==> 1+(sum [2,3])
==> 1+(2+(sum [3]))
==> 1+(2+(3+(sum [ ])))
==> 1+(2+(3+0))
==> 1+(2+3)
==> 1+5
==> 6
```

Computation of `(1+(2+3))` cannot begin until the final tail recursive call `(3+0)` is completed.

Either applicative or iterative techniques can be used to eliminate tail recursion. Both rewrite the function so it includes a new parameter called the *accumulating parameter* which accumulates operations such as sums at the beginning (not the end) of recursive calls. Examples of applicative and iterative tail recursion elimination are provided in the following paragraphs.

Applicative Tail Recursion Elimination It may be desirable to eliminate tail recursion by translating the tail recursive function to applicative rather than imperative form (e.g., so that the entire translator can be written in an applicative language).

One applicative solution involves rewriting the tail recursive function to include an accumulating parameter that stores intermediate function results. For example, the accumulating parameter *acc* is included in `sum1` below (again, `n` is a number and `ns` is a list of numbers):

```
sum1 acc [ ]    = acc
  '  acc (n:ns) = (sum1 (acc + n) ns)
```

`Sum1` can be invoked by the following function call to `sum` (where `L` is a list):

```
sum L = (sum1 0 L)
```

The revised call to `sum [1,2,3]` is evaluated faster and in constant space as follows:

```
    sum [1,2,3]
==> (sum1 0 [1,2,3])
==> (sum1 (0+1) [2,3])
==> (sum1 1 [2,3])
==> (sum1 (1+2) [3])
==> (sum1 3 [3])
==> (sum1 (3+3) [ ])
==> (sum1 6 [ ])
==> 6
```

Notice that `sum1` is applied at the beginning of each reduction step, and the accumulating parameter holds intermediate sums. Step-by-step calculation of the intermediate sums is forced by eager evaluation but not by lazy evaluation.

Iterative Tail Recursion Elimination Another imperative method to eliminate tail recursion replaces the tail recursive call with a goto statement that transfers control to the beginning of the tail recursive code without initiating a new activation record. An accumulating parameter located inside the iterative loop holds results of the required operations [Davie, 1992]. For example, the function `sum` described above could be transformed to the iterative function `sum2` below (A C-like syntax is used where `acc` is the accumulating parameter, `L` is a list, `nil (L)` is true if the list is empty and false otherwise, `head(L)` is the head of list `L`, and `tail (L)` is the tail of list `L`):

```
int sum2 (L);
list:L;
int acc = 0;
   {
top:  if nil (L) return acc;
         else acc = acc + head (L);
      L = tail (L);
      go to top;
   }
```

Then `sum = sum2 [1,2,3]` would compute `6` iteratively with no recursive calls.

3.2 Memoization

Since referential transparency (§ 1.2) guarantees that function application cannot cause side effects, a function with a specific set of arguments always reduces to the same normal form (§ 1.4.2), regardless of the number of times it is evaluated. Rather than revaluating equivalent expressions, *memoization* changes the form of a function so it can reuse previously computed results [Michie, 1968]. The memoized function acts like a memo pad that can be consulted to recall previous computations.

If multiple instances of a function occur inside the same expression, detection of the equivalence is easy. For example, a compiler can detect that the two instances of `(f 4)` in:

```
(f 4) + (f 7) * ( f 4)
```

are equivalent and emit code to evaluate `(f 4)` only once. However, normal compilers would fail to detect equivalence if the equivalent instances were to occur in different expressions, particularly if they are widely separated in the program.

Similarly, equivalent functions applied during recursive expansion could go undetected. For example, consider the following Fibonacci function example:

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

Partial expansion of `(fib 5)` is shown below:

```
    (fib 5)
 ⇒  (fib 4) + (fib 3)
 ⇒  (fib 3) + (fib 2) + (fib 3)
 ⇒  ...
```

Notice that the expansion creates two instances of `(fib 3)`. Since they are created by different invocations of `fib`, their equivalence would normally go undetected and each instance would cause evaluation or reevaluation of `(fib 3)`. Revaluation of expressions can be disastrous. In `fib`, it causes time complexity to grow exponentially with the size of `n`.

The goal of memoization is to detect *all* equivalent function applications in a program. The first evaluation causes the function's value to be stored in a temporary data area. This is referred to as *caching*. Subsequent equivalent applications access the cache for the value avoiding recomputation. A function whose values are saved is called a *cached function* and the results that are saved are called *cached values*. A *directory* is used to record where cached values are stored.

For example, a programmer might identify Fibonacci as a cached function as follows [Hudak, 1989]:

```
cached memo fib
fib 0 = 0
  ' 1 = 1
  ' n = (fib (n-1)) + (fib (n-2))
```

As is the case with tail recursion elimination (§ 3.1), either imperative or applicative techniques can be used to perform memoization. Use of the imperative techniques is generally more powerful, but requires a shift from applicative to imperative execution in order to handle memo functions. Imperative and applicative memoization techniques are described below.

Imperative Memo Functions A translator could convert the applicative function to an imperative function called a *memo function*. For example `fib` could be converted to an imperative memo function `cache_fib` using the C-like syntax shown below:

```
int cache_fib (i);
int i;
{
  int r;
  if present (i,dir)
      return value (i,dir);
  r = if (i<2) 1
      else (cache_fib (i-1)) +
                (cache_fib (i-2));
  insert (r,i,dir);
  return r;
}
```

where the following definitions apply:

- `dir`—is a directory data structure (e.g., array or linked-list).
- `present (i,dir)`—is a function that returns true if a previous invocation of `(cache_fib i)` has stored its value in `dir` at index position `i`; otherwise, it returns false.
- `value (i,dir)`—is a function that returns the value stored in `dir` at position `i`.
- `insert (r,i,dir)`—is a function that inserts the value `r` at position `i` in directory `dir`.

`Cache_fib` behaves as follows. The function `present` examines the directory to see if results for the current function call have already been calculated. If so, `value` accesses that result directly from the directory. If not, the computation is initiated. When complete, `insert` stores the computed value in the directory so it is available when future function calls are invoked.

Applicative Memo Functions As was established earlier, a problem with imperative memo functions is that they require translation of an applicative function definition to imperative format. It may be preferable to use applicative memo functions (e.g., if the entire translator must be written in an applicative language).

The method outlined below can be used to construct applicative memo functions. It was originally proposed by Keller and Sleep [1986] and was adapted to Haskell by Hudak in [1989].

Functional languages often use data structures called *streams* to represent infinite ordered lists of values (or rather semi-infinite lists, because nothing can be infinite in the finite space of a digital computer). These streams can be viewed as semi-infinite arrays where the integers $\geq$ `0` (e.g., `0, 1, 2, 3, ...`) are subscripts of the array. Eager evaluation of a stream fails to terminate, but lazy or "demand" evaluation returns stream elements one at a time.

If the domain of a function is of integer type $\geq$ `0`, a stream can be used as the function's directory. For example, if the function is `(fib n)` and `dir` is a stream representing the function's directory, the value of `(fib i)` for `0` $\leq$ `i` $\leq$ `max` would be stored in the `i`th element of stream `dir`.

In the example that follows, assume the function uses constructs called *suicidal suspensions* [Friedman and Wise, 1976]. That means when a function is evaluated for the first time, its value is automatically inserted in directory `dir`. Since the value in the directory will be accessed on subsequent calls instead of re-evaluating the function, the expression is said to "commit suicide."

During reduction the cached Fibonacci function:

```
cached memo fib
fib 0 = 0
  ' 1 = 1
  ' n = (fib (n-1)) + (fib (n-2))
```

could be translated to:

```
fib = (cache dir)
where
  dir 0 = 1
    '  1 = 1
    '  n = (fib (n-1)) + (fib (n-2))
```

This causes `fib` to invoke `cache` each time `fib` is accessed. It is important to note that `cache` can be any user-defined function that achieves the implementor's desired results. For example, `cache` is implemented as an array below (the array has similar behavior characteristics to streams introduced above).

```
cache dir = \n -> (array(0,max)
  [(i,(dir i))| i <- [0..max]]) ! n
```

Incorporating this `cache` definition yields the following applicative memo function for `fib`:

```
fib n = (array (0,max)
 [(i,(dir i)) | i <- [0..max]) ! n
where dir 0 = 1
       '  1 = 1
       '  n = (fib (n-1))+(fib (n-2))
```

The array `dir` is instantiated only once and that instantiation is used for each invocation of `fib`. Therefore, the same directory is used for each access to `fib`. The first access to `(fib i)` for $0 \leq i \leq max$ causes `dir` to compute its value. When evaluation is complete, the computed value is automatically stored in `dir` at location `i`. Subsequent accesses to `(fib i)`, draws the value directly from the array (i.e., recomputation is unnecessary). Then the accessing expression commits suicide.

Directory Structures Three cache storage schemes can be implemented using either applicative ("purely" functional) or imperative techniques. A *linked-list cache* (where domain elements are accessed by a list search using an equality test) is able to handle functions with infinite domains, but the access times grow linearly with the size of the list.

In contrast, a *tuple cache* stores cached values in a fixed-sized contiguous space and accesses domain elements by index. This places a limit on the size of the function domain but function values can be accessed in constant time. It can also make spacing between range values sparse, but it does not limit the size of the range. Tuple cache also requires that the tuples be mapped to integers. This is not always possible with strings and other structures.

A tree organization referred to as *tries cache* is able to represent infinite domains like the linked-list cache, but the domain elements must have an order (e.g., they are suitable for comparison with a relational operator) and it consumes slightly more space (one pointer per linked-list node and two per tries node). It has a respectable $\log_2(\mathit{directory\_size})$ access time when the tree is balanced.

Tuple cache storage schemes require that the demanded function arguments be sparse with respect to space allocated to the function directory. This is called the *sparse matrix problem* and leads to unused (wasted) space in the directory. A solution is to reorganize the directory to minimize unused space. This is called an *arbitrary* argument structure because entries correspond to arbitrary arguments. Unfortunately, methods to implement arbitrary structures require extensive analysis that has rendered them impractical so far [Haines and Böhm, 1991].

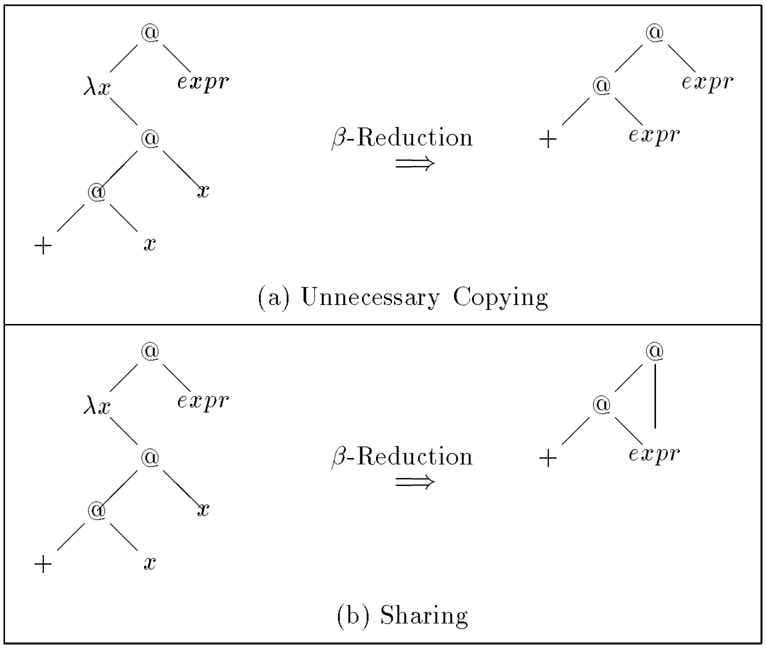


Figure 14: λ-Argument Copying

Analysis of Memoization Memoization is necessary to invoke sharing across function calls and improve repetition performance when recursion is used. However it suffers from the following problems:

- Caching adds a significant amount of administration to the original implementation design.
- No single cache storage scheme is best for all situations; it is difficult to evaluate the current conditions and invoke the appropriate method.
- It may be necessary to periodically purge large unused data structures or reduce them to a more manageable size to conserve storage space.
- Since the directory is a shared structure that is both read from and written to, access to the same directory element must be limited to one process at a time. These access restrictions can cause bottlenecks and deadlock if interrupts are allowed. The deadlocks must be detected and cleared.

3.3 λ-Argument Copying

During β-reduction, an actual argument is substituted for the formal parameter, wherever the formal parameter variable appears in the body of the λ-expression (§ 2.4). This substitution is typically accomplished by copying, and may involve unnecessary copying if there are multiple instances of the formal parameter variable in the λ-body. For example, when the λ-expression

$$(\lambda x. + x\ x)\ expr$$

is β-reduced as shown in figure 14 (a), its argument $expr$ is copied twice. Furthermore, $expr$ must be evaluated twice during the ensuing δ-reduction, duplicating work.

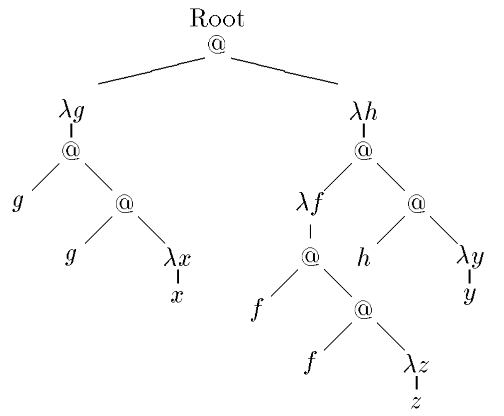


Figure 15: Graphical Representation of $(\lambda g.g\ g(\lambda x.\ x))(\lambda h.((\lambda f.f\ f(\lambda z.z))h(\lambda y.y)))$

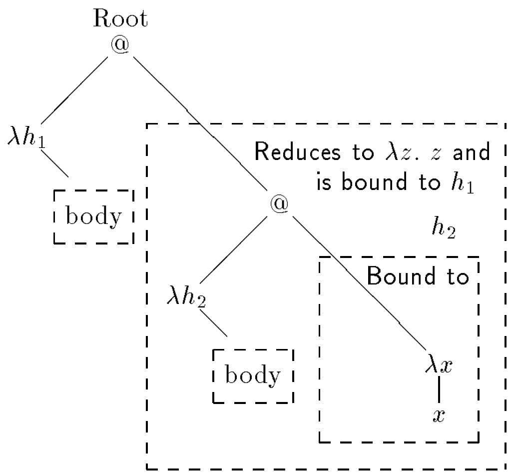


Figure 16: Dual Bindings for λh

Copying may be avoided if it is possible to share a single instance of *expr* as shown in figure 14 (b). However, sharing is complicated if *expr* contains free variables, because it will evaluate differently depending on the bindings of those free variables.

If free variables are present, sometimes a prudent evaluation of the expression will minimize copying. For example, eager evaluation of *expr* would reduce it to WHNF (§ 1.4.2) prior to the β-reduction, possibly reducing the size of the copied data structure.

However, Lamping [1990] notes some λ-expressions require copying regardless of the evaluation method used. For example,

$$(\lambda g.g\ g(\lambda x.x))(\lambda h.\ ((\lambda f.f\ f(\lambda z.z))h(\lambda y.y)))$$

(graphically represented in figure 15) has two redexes, λg (outer—reduced first using lazy evaluation) and λf (inner—reduced first using eager evaluation).

If the outer redex is evaluated first by performing a β-reduction (§ 1.4.3) on λg, its argument, $(\lambda h\ \ldots)$, which includes the inner redex will be applied to the dual instances of g in the λg body as shown in figure 16. Sharing similar to figure 14 (b) is not possible, because each λh is bound to a different argument. Figure 16 demonstrates that the rightmost λh (λh_2) is bound to the argument $\lambda x.\ x$, while the leftmost (λh_1) is bound to the expression that λh_2 reduces to (which is $\lambda z.\ z$).

If the inner redex is evaluated first, the same problem is encountered. Eventually, there will be two different bindings for λh, so the dual instances of f in the body of λf cannot be shared.

However, in both cases it is only variables that are duplicated and receive different bindings that cannot be shared. Duplicate variables, all of which receive the same bindings, can be shared.

Lamping seeks to exploit these extra sharing opportunities and avoid duplication of work. To do this he adds a new control structure to the graph that indicates partial sharing. For example, the sharing of the variables f and g in figure 15 is represented by the triangular *fan-in* control structure shown in figure 17 (where the $\star$ path connects f and g to the leftmost parent and $\circ$ connects them to the rightmost parent). The β-reduction of λg replaces g with λh (and its associated structure) as shown in figure 18.

Next, λh is moved to the top of the fan-in

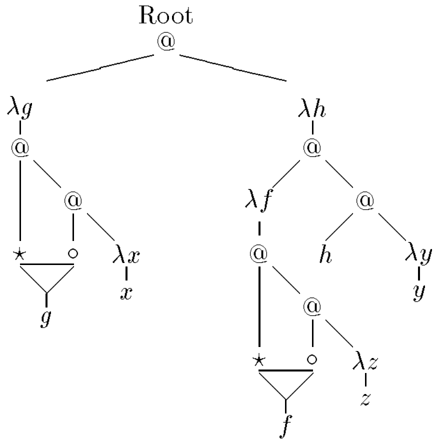


Figure 17: Control Structures for f and g

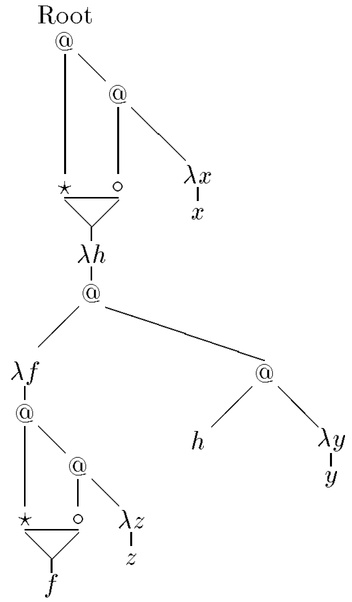


Figure 18: β-Reduction of λg

control structure and is separated into two parts (λh and $\lambda h'$) to reflect the different bindings. Occurrences of h in the body of λh are also separated (into h and h') with a *fan-out* control structure as shown in figure 19. Notice that most of the graph is shared. The control structure indicates there are two bindings for λh (λh and $\lambda h'$) and the fan-out structure shows that λh is bound to h and $\lambda h'$ is bound to h'.

To implement reduction, Lamping introduces a list of rewrite rules that cover all possible reduction situations. At any time, several different rewrite rule applications may be present in a graph, and due to referential transparency, they can be reduced in any order. Therefore, the algorithm is appropriate for both sequential and parallel implementations.

This technique is sensible and appears simple on the surface, but it can become quite complex. For example, the expression in figure 15 contains two shared expressions, g and f. During reduction, each can generate a number of fan-in control structures, each of which must be associated with a unique fan-out structure. The more sharing there is, the more complex is the control structure. Furthermore, pattern matching is required to identify where the rewrite rules can be applied. This can be expensive, and the technique does not even identify all sharable expressions in a graph. Other techniques such as applicative caching/memoization are necessary to do that, adding to already abundant overhead costs.

The matter of complexity raises the question of whether algorithms such as this one are truly an improvement over more limited sharing algorithms. Storage and garbage collection requirements seem to be reduced because less copying is required, but overhead (e.g., to support pattern matching) undoubtedly reduces these gains. It is even unclear whether methods such as this one execute faster. Increased complexity certainly raises a specter of doubt. Unfortunately, performance comparisons are not available to allay those doubts.

3.4 Partial Evaluation

Partial evaluation reduces a functional program using static compile-time inputs to a *residual program* that is intended to have

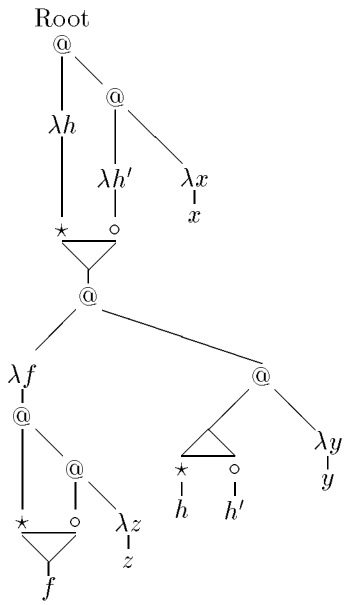


Figure 19: Fan-In and Fan-Out for λh

more desirable run-time performance than the original program. For example, assume a program p requires several inputs to complete execution, and some of those inputs, s, can be applied statically, while the remaining inputs, d, must be applied dynamically. Applying s to p produces a residual program r, or $p\ s = r$, where r fulfills $r\ d = p\ s\ d$.

When r is run with a series of dynamic inputs $d_1, d_2, \ldots, d_n$, more efficient execution results can be achieved than if each of $p\ s\ d_1, p\ s\ d_2, \ldots, p\ s\ d_n$ is run independently instead. If the execution of p happens to repeat the static computations, there can be a reduction in execution time, even if r is applied only once [Holst and Gomard, 1991; Takano, 1991].

For example, the *Ackerman* program:

```
ack m n = if m == 0 then n+1
   elseif n == 0 then (ack (m-1) 1)
      else(ack (m-1) (ack m (n-1)))
```

partially evaluated by applying the static value 2 to `m` generates the following residual program:

```
ack2 n = if n == 0 then (ack1 1)
      else (ack1 (ack2 (n-1)))
ack1 n = if n == 0 then (ack0 1)
      else (ack0 (ack1 (n-1)))
ack0 n = n+1
```

Unfortunately, residual programs are only valid for the specific static inputs that are selected. If other static inputs are desired, a new residual program must be produced. Also, either the user or the compiler must generate the residual program. If the user does it, effort is diverted away from the problem solving task and if the compiler does it, translation complexity is increased. Perhaps the biggest problem, however, is that, if lazy evaluation is used, execution speed is slow and if eager evaluation is used, the partially evaluated program may not terminate.

3.5 Mixed-Order Evaluation

There are good reasons to use both eager and lazy evaluation [Field, 1990]. This type of evaluation is referred to as *hybrid* or *mixed-order evaluation*. John Field notes that hybrid schemes use the least amount of steps yet still allow component sharing.

For example, given that I is the identity function, consider $\mathsf{N} \equiv \mathsf{N}_1\mathsf{N}_2$, where $\mathsf{N}_1 \equiv \lambda x.\ (x\ w)\ (x\ z)$ and $\mathsf{N}_2 \equiv \lambda y.\ (\mathsf{I}\ y)$. Figure 20 (a) uses purely normal order or lazy evaluation, whereas figure 20 (b) uses mixed-order evaluation.

Notice that mixed-order evaluation uses one less reduction step in figure 20. The difference is that the shared argument $\lambda y.\ (\mathsf{I}\ y)$ is applied lazily in the second step of the lazy evaluation example, whereas it is reduced eagerly in the second step of the mixed-order evaluation example.

Consequently, mixed-order evaluation can be used to improve performance. Some methods used to invoke mixed-order evaluation are listed below.

User Annotations Hudak suggests using a language implementation that is lazy, ex-

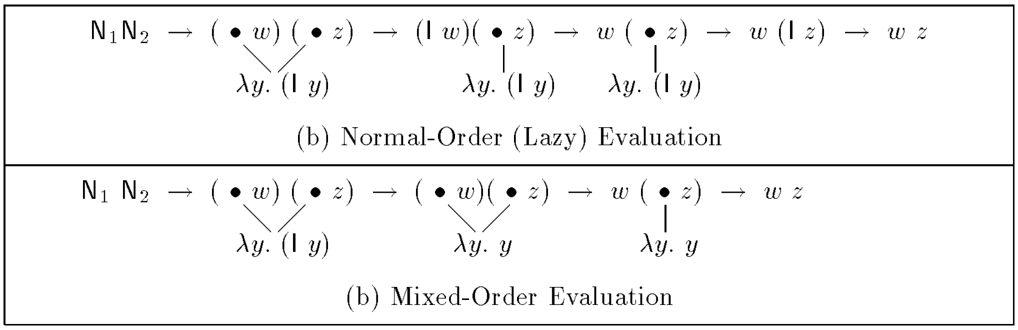


Figure 20: Evaluation of Shared Expressions

cept the syntax allows users to mark (annotate) expressions with a hasher `(#)` to invoke eager evaluation [Hudak, 1986a; Hudak, 1986b]. His rationale is that educated users can make prudent decisions on which expressions to annotate. Some proponents of user annotations do not feel that compilers are capable of making the same informed decisions.

Compiler Annotations In contrast, Burton chooses to annotate intermediate-code application nodes as either $@_L$ for lazy, or $@_E$ for eager [Burton, 1984]. Their assignment at the intermediate-code level implies the compiler will insert the annotations, but no formal method to effect those decisions is included in the description.

Burton's seeks to control the size of data structures by prudently labeling application nodes. For example, if there are no other assignment considerations and eager evaluation will terminate (i.e. is strict) *and* reduce the size of an application node's data structure, label the node as eager. If there are no other considerations, and eager evaluation will *not* terminate (i.e. is not strict) or it will increase the size of the application node's data structure, label the node as lazy. Burton calls this evaluation scheme *Mixed-Order Evaluation* to distinguish it from Normal- and Applicative-Order Evaluation.

Strictness Analysis It is possible to determine which application nodes can safely be reduced eagerly through a procedure called *strictness analysis.* C. D. Clack, Simon Peyton Jones and others have proposed techniques that allow compilers to identify and annotate strict functions and function arguments [Clack and Peyton Jones, 1985b; Peyton Jones, 1987]. As a consequence, strict function arguments can be evaluated eagerly without violating lazy semantics. This can dramatically improve the performance of lazy implementations [Langendoen, 1993].

A function that always requires an argument is said to be *strict* on that argument (§ 1.3). More formally, assume a function $\mathtt{f}$ has $\mathtt{n}$ arguments $\mathtt{x}_1$, $\mathtt{x}_2$, ..., $\mathtt{x}_n$, and $\mathtt{x}_i$ is one of them (where $1 \leq i \leq n$). Function $\mathtt{f}$ is strict on argument $\mathtt{x}_i$ iff:

$$(\mathtt{f}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_{i-1}\ \perp\ \mathtt{x}_{i+1}\ \ldots\ \mathtt{x}_n) \equiv \perp$$

where $\perp$ is a non-terminating expression. A function that is strict on all of its arguments is said to be a *strict function.*

The strictness analysis method in [Peyton Jones, 1987] assumes that each argument $\mathtt{x}_i$ is in turn $\perp$ (while the other arguments are *assumed* to terminate). In each case, if $\mathtt{f}$ can be shown to be $\perp$ then $\mathtt{f}$ is strict on that argument and the argument can safely be evaluated eagerly. If not, $\mathtt{f}$ is not strict on that argument and the argument should be evaluated lazily.

Strictness can be computed by converting functions to Boolean expressions that evaluate to `false` if the functions are strict for a given set of argument inputs or `true` if they are not. For example, the following algorithm is adapted from [Peyton Jones, 1987].

Given an argument of the function

$$(\texttt{f}\ \texttt{a}_1\ \texttt{a}_2\ \ldots\ \texttt{a}_n)\ \equiv\ \texttt{expr}$$

for $n > 0$, the following determines whether `f` is strict on that argument (assume $\vee$ is logical *or* and $\wedge$ is logical *and*):

1. Select an argument $\texttt{a}_i$ where $0 < i \leq n$ and assume it does not terminate. Non-termination is represented by the value `false` (e.g., $\texttt{a}_i$ `= false`).

2. All other arguments may or may not terminate, but to ensure the termination of `f` is dependent only on $\texttt{a}_i$, assume they do terminate, where termination is represented by the value `true` (e.g., $\texttt{a}_j$ `= true` where $0 < j \leq n$ and $i \neq j$).

3. Apply the Boolean values to all arguments in sub-expressions of `expr` as follows:

   (a) A constant sub-expression always terminates so it is assigned the value `true`.

   (b) Any sub-expression of the form $\texttt{e}_1$ `op` $\texttt{e}_2$ (where `op` is a binary arithmetic operator) terminates if $\texttt{e}_1$ or $\texttt{e}_2$ terminates (i.e., if $\texttt{e}_1 \wedge \texttt{e}_2$ terminates).

   (c) Any sub-expression of the form `if` $\texttt{e}_1$ `then` $\texttt{e}_2$ `else` $\texttt{e}_3$ terminates if $\texttt{e}_1$ terminates and $\texttt{e}_2$ or $\texttt{e}_3$ terminates (i.e., if $\texttt{e}_1 \wedge (\texttt{e}_2 \vee \texttt{e}_3)$ terminates).

As a practical example consider the function:

```
g p q r = if p then (q+r) else (q+p)
```

Assume `p` fails to terminate but `q` and `r` do terminate (e.g., `p = false`, `q = true`, and `r = true`). Application of the algorithm rules transforms `g` to:

$$
\begin{array}{ll}
 & \texttt{g p q r} \\
\Rightarrow & (\texttt{p} \wedge \texttt{true}) \wedge ((\texttt{q} \wedge \texttt{r}) \vee (\texttt{q} \wedge \texttt{p})) \\
\Rightarrow & \texttt{p} \wedge (\texttt{q} \wedge (\texttt{p} \vee \texttt{r})
\end{array}
$$

Now, substituting `p = false`, `q = true`, and `r = true` yields:

$$
\begin{array}{ll}
 & \texttt{g false true true} \\
\Rightarrow & \texttt{false} \wedge (\texttt{true} \wedge (\texttt{false} \vee \texttt{true})) \\
\Rightarrow & \texttt{false}
\end{array}
$$

Therefore, when `p` does not terminate, `g` also does not terminate, so `g` is strict on `p`. Similarly:

$$
\begin{array}{l}
\texttt{g true false true} \Rightarrow \texttt{false} \\
\texttt{g true true false} \Rightarrow \texttt{true}
\end{array}
$$

Therefore, `g` is strict on `q`, but `g` is *not strict* on `r`.

This method assumes that the function contains no free variables (§ 1.4.2). For that reason, it works well when applied to lambda lifted expressions such as super-combinators.

Complications also arise when strictness analysis is applied to recursive functions since at any point during evaluation information may be required that is currently being computed [Langendoen, 1993; Peyton Jones, 1987; Hughes, 1990]. One solution is to compute successive approximations incorporating more refined information at each step. For example, at first all arguments are assumed to be non-strict. This information is propagated through the tree and yields a subset of the arguments that are strict. The second traversal yields more strict arguments, and so on until all strict arguments have been detected. This limit, often called the fixed point, is very time consuming to determine.

Although the above techniques determine which arguments to evaluate, further compiler analysis can be performed to determine just how far to evaluate the argument based on the function applied and the data-structure of the argument. For example, if the function is empty and the strict argument is a list, there is no need to evaluate individual components of the list. On the other hand, if the function is performs summation, all members of the list must be reduced to values before the function is applied. The *evaluation transformer* model takes the function and data-structure of list type arguments into consideration during strictness analysis [Langendoen, 1993; Burn, 1991].

Multitudes of other strictness analysis schemes have been proposed. For example, refer to [Mycroft, 1981; Hall and Wise, 1987; Nielson, 1987; Kuo and Mishra, 1987; Wadler, 1988].

In some cases, fully eager evaluation is a better alternative than fully lazy evaluation and vice versa. Mixed-order schemes can capitalize on the advantages of each and avoid their disadvantages. Most mixed-order evaluation schemes involve an implementation that is lazy except strict expressions are marked (or annotated) for eager evaluation. Hudak [1986a; 1986b] allows the programmer to specify those annotations and thus control the evaluation, whereas, Burton [1984], Clack and Jones [1985b; 1987] apply compiler annotations transparent to the programmer.

4 Parallel Implementation Techniques

Because of referential transparency (§ 1.2), the evaluation of a reducible expression (redex) does not interfere with the evaluation of any other redex. Since they do not interfere with one another, redexes may be evaluated in parallel. This is desirable since it implies that no new language constructs must be added to enforce synchronization [Peyton Jones and Lester, 1992]. Implementations that can in fact parallelize these independent components of work could achieve very high levels of performance. For that reason, parallel implementations are being actively researched.

Much of the information presented in earlier sections of this survey also applies to parallel implementations. For example, many parallel implementations use an intermediate representation based on lambda calculus. It is also likely that one of the sequential evaluation methods (§ 2) will be applied to individual processors in a multiprocessor system, and that one or more of the optimization techniques (§ 3) will be used to improve multiprocessor performance.

However, parallelism introduces some problems which are very different from those encountered in sequential implementations. The remainder of this section is devoted to describing methods used to invoke parallel tasks (§ 4.1); demonstrating how the parallelism can be viewed as either conservative or speculative in nature (§ 4.2); describing various parallel memory organizations (§ 4.3); identifying problems caused by the blocking and resumption of parallel tasks (§ 4.4); analyzing task and data distribution issues (§ 4.5), investigating the implications of task size in parallel implementations (§ 4.6); discussing methods used to reclaim garbage (§ 4.7); accessing data as aggregates rather as individual components (§ 4.8); and applying vectorization to program components (§ 4.9).

Wherever possible, the techniques are correlated with the features found in actual implementations. The implementations are Applicative Language Idealized Computing Engine (ALICE) [Darlington and Reeve, 1981], Abstract Machine for Parallel Graph Reduction (AMPGR) [George, 1989], Flagship [Watson and Watson, 1986; Watson and Watson, 1987; Banach *et al.*, 1988], GAML [Maranget, 1991], Graph Reduction in Parallel (GRIP) [Peyton Jones *et al.*, 1987; Clack and Peyton Jones, 1985a; Hammond and Peyton Jones, 1991], Highly Distributed Graph-Reduction (HDG) [Kingdon *et al.*, 1991], Parallel Experimental Reduction Machine (HyperM) [Barendregt *et al.*, 1992], Parallel ABC (PABC) [Plasmeijer and van Eekelen, 1993; Nöcker *et al.*, 1991], Parallel Abstract Machine (PAM) [Loogen *et al.*, 1989], Parallel Functional Language (ParAlfl) [Hudak, 1986a; Hudak, 1986b; Hudak, 1989], Parallel SML [George and Lindstrom, 1992], Qlisp [Goldman and Gabriel, 1988; Goldman and Gabriel, 1989], Streams and Iterations in a Single Assignment Language (SISAL) [Böhm *et al.*, 1991], and $<\nu,\mathrm{G}>$ [Augustsson and Johnsson, 1989].

4.1 Invoking Parallelism

While parallelism is implicit in functional languages, not all of it is useful parallelism. There are several ways to determine which tasks are invoked in parallel. Each method handles the problem of useful tasks differently. For example, *random* implementations blindly select tasks without regard to their usefulness; *explicitly annotated* implementations require the programmer to designate parallel tasks by annotating them, again without regard to their usefulness; current *implicitly annotated* systems use compiler generated strictness annotations (§ 3.5) to invoke only useful tasks; and *structure oriented* systems rely on program structure to determine parallel tasks, not all of which are useful. Each of these alternatives is described in greater detail below.

Random Parallelism If the invocation of parallelism is dependent on a random selection process such as picking arbitrary components from a pool of tasks, that parallelism is referred to as *random parallelism.*

ALICE and Flagship both invoke parallelism randomly. They represent program graph nodes as packets and then processors select these packets arbitrarily from a pool.

Random parallelism has the advantage of being simple, but lacks the discipline of organized reduction methods and therefore can be very inefficient.

Explicit Annotations Explicit annotations are special constructs added to language syntax that allow programmers to invoke parallelism. For example, Parallel SML invokes parallel tasks with the *spark* primitive. The statement:

`spark` *expression*

invokes the expression on an available processor. GAML, PAM, PABC, ParAlfl, Qlisp, and $<\nu,\mathrm{G}>$ use similar explicit annotations to invoke parallel tasks.

ParAlfl's annotations include the capability to specify processor numbers. The statement:

expression `on proc n`

invokes the expression on processor number `n`.

Qlisp includes a *propositional parameter* in its annotations that allows parallelism to be either invoked or suppressed. For example:

`spawn` *prop expression*

invokes the expression in parallel with the current task if prop evaluates to `true` (nonzero) or evaluates the expression on the same processor as the current task if prop evaluates to `false` (zero).

Qlet is another means of explicitly invoking parallel tasks in Qlisp. It has the form:

`qlet` *prop* `((`$\mathtt{x}_1$ arg_1`)`... `(`$\mathtt{x}_n$ arg_n`)).`*body*

In this statement, $\arg_1 \ldots \arg_n$ are invoked on separate processors if prop evaluates to `true` or they are all evaluated on the same processor as the current task if prop evaluates to `nil` (i.e., `qlet` behaves like a normal `let` expression).

If prop evaluates to the special symbol `eager`, $\arg_1 \ldots \arg_n$ are invoked on separate processors and the body of the `qlet` is also invoked in parallel. This means that if the body requires any unevaluated arguments during execution, it must block until they are evaluated.

An advantage of explicit annotations is that the programmer maintains control over processor activities. The primary disadvantage is that the programmer must worry about parallelism, so parallelism is not transparent to the programmer.

Implicit Annotations Implicit annotations are applied to the program intermediate code by the compiler. Recall that Burton annotates intermediate code application nodes as either eager, $@_E$, or lazy, $@_L$ (§ 3.5). Usually, an application node's left child is a function and the right child is the argument, so $@_E$ means evaluate the argument eagerly and $@_L$ means evaluate the argument lazily. Burton also applies a parallel annotation, $@_P$, that behaves in the same manner as $@_E$ except the argument in $@_P$ is sparked immediately on another processor [Burton, 1984].

A strictness analysis technique proposed by C.D. Clack and Simon Peyton Jones uses the compiler to identify strict arguments and annotate them for eager evaluation (§ 3.5). In a parallel system, when the annotation is encountered, the argument can be transported to another processor for evaluation in parallel.

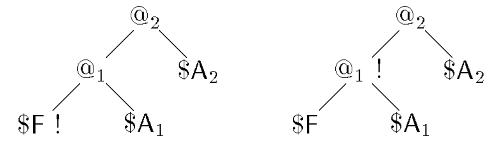
Strict arguments sometimes annotated with an exclamation point. Actually, [Peyton Jones, 1987] suggests applying strictness annotations to function nodes, argument nodes, or both depending on the situation. For example, in figure 21 (a) the function node $\$\mathsf{F}$ in the super-combinator:

$$\$\mathsf{F}\ \$\mathsf{A}_1\ \$\mathsf{A}_2$$

is annotated, while in figure 21 (b) $@_1$ (the application node associated with argument $\$\mathsf{F}$'s first argument $\$\mathsf{A}_1$) is annotated. Although both annotations indicate that $\$\mathsf{F}$ is strict on $\$\mathsf{A}_1$, $\$\mathsf{F}!$ implies that $\$\mathsf{F}$ is strict on all arguments (e.g., $\$\mathsf{A}_1$ and $\$\mathsf{A}_2$), while $@_1!$ indicates $\$\mathsf{F}$ is only strict on $\$\mathsf{A}_1$.

Both function and argument annotations are useful. For example, in the expression

$$(\mathsf{if}\ \$\mathsf{C}\ \$\mathsf{T}\ \$\mathsf{F})\ \$\mathsf{A}$$

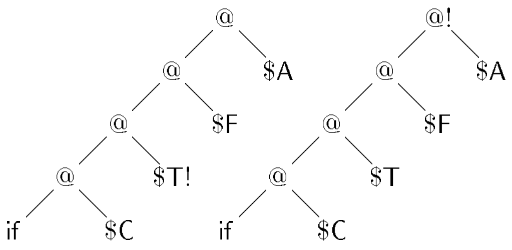


(a) Function (b) Argument

Figure 21: Annotations for $\$\mathsf{F}\ \$\mathsf{A}_1\ \$\mathsf{A}_2$

if $\$\mathsf{T}$ is strict and $\$\mathsf{F}$ is not, there is no way to annotate the application node for argument $\$\mathsf{A}$ correctly. If the condition $\$\mathsf{C}$ is *true*, $\$\mathsf{A}$ can be evaluated in parallel with the if function. If $\$\mathsf{C}$ is *false*, it cannot be. The solution here is to annotate the $\$\mathsf{T}!$ function node as shown in figure 22 (a). The function reduces to ($\$\mathsf{T}!$) $\$\mathsf{A}$ if $\$\mathsf{C}$ evaluates to *true*, or ($\$\mathsf{F}$) $\$\mathsf{A}$ if $\$\mathsf{C}$ is *false*.

Conversely, if $\$\mathsf{T}$ and $\$\mathsf{F}$ are both strict, annotation of $\$\mathsf{A}$'s application node as shown in figure 22 (b) is useful. It invokes evaluation of the argument $\$\mathsf{A}$ in parallel with the if statement.



(a) Function (b) Argument

Figure 22: Annotations for (if $\$\mathsf{C}\ \$\mathsf{T}\ \$\mathsf{F}$) $\$\mathsf{A}$

AMPGR, GRIP, HDG, and SISAL all invoke parallelism with implicit annotations.

Implicit annotations require more complex compilers than explicit annotations, but the programmer's task is easier. Parallelism is transparent to the programmer.

Structure-Oriented Parallelism can also be derived from a program's structure. For example, if the program uses a divide-and-conquer algorithm, whenever division takes place, new parallel tasks can be invoked. This is referred to as *structure-oriented* parallelism. However, only HyperM uses this approach to parallelism.

Structure-oriented parallelism is reasonably simple to implement. However, it can only be applied to programs with the requisite structure. If the program's structure is not appropriate, it must be transformed to the required structure. This complicates the programming process and can adversely effect reliability.

Figure 23 summarizes methods used by current functional programming language implementations invoke parallelism.

Parallelism	Implementation
Random	ALICE
	Flagship
Explicitly Annotated systems	Parallel SML
	ParAlfl
	$<\nu,G>$
	GAML
	PAM
	PABC
	Qlisp
Implicitly Annotated	AMPGR
	GRIP
	HDG
	SISAL
Structure-Oriented	HyperM

Figure 23: Invocation of Parallelism

4.2 Conservative and Speculative Parallelism

The process of making reduction expressions (redexes) available to a processor is called *sparking* tasks. The term spark is intended to convey the image of a lighted match touching a redex. The 'fire that the match causes' (e.g., execution of the redex) spreads when execution invokes (sparks) other redexes. At program start, the program's root redex is sparked, and then as other redexes become executable and there are processors to accommodate them, they are sparked too.

Conservative Parallelism The sparking of tasks only when it is certain they will be needed is referred to as *conservative parallelism*. The effect of conservative parallelism is similar to lazy evaluation in sequential implementations. Both invoke tasks only when they are needed. For example, consider the function:

$$\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \tag{4}$$

where e_1 is a conditional (Boolean) expression, e_2 is executed on a true condition, and e_3 is executed on a false condition. Only one of e_2 and e_3 needs to be evaluated. Which one is not known until e_1 is evaluated. Conservative parallelism holds off sparking e_2 and e_3 until e_1 has terminated, and then sparks only one of them. Unless it is shared, the other expression is discarded.

The problem with conservative parallelism is that some delayed tasks eventually are needed, either because evaluation of the current function or another sharing function sparks them. When a need for them is established, late evaluation must be initiated.

Speculative Parallelism The sparking of tasks when they may or may not be needed is referred to as *speculative parallelism*. The effect of speculative parallelism is similar to eager evaluation in sequential implementations. Both invoke tasks that may not be needed. For example, in expression 4 above, speculative parallelism would spark e_1, e_2, *and* e_3 in parallel, so all three could be in reduced form when they are needed by the if statement. Unfortunately, sometimes speculative parallelism evaluates unneeded tasks. If the unneeded task is non-terminating, it would cause unnecessary program non-termination.

Comparison of Conservative and Speculative Parallelism The effects of conser-

Argument	Eager Evaluation	Lazy Evaluation
Evaluation required	Conservative parallelism No wasted effort	Overly conservative parallelism No wasted effort
Evaluation might be required	Speculative parallelism Possibly wasted effort	Possibly overly conservative parallelism No wasted effort
Evaluation not required	Wasted parallelism Wasted effort	No parallelism No wasted effort

Figure 24: Speculative versus Conservative Parallelism

vative and speculative parallelism are shown in figure 24. As the figure indicates, if it can be determined that the evaluation of arguments is required, both eager and lazy evaluation invoke conservative parallelism with no wasted effort. However, lazy evaluation is overly conservative. It may delay some tasks and therefore bypass parallelism opportunities.

When a technique such as compile-time strictness analysis is *unable* to determine whether or not the evaluation of arguments is required, eager evaluation invokes speculative parallelism on them and lazy evaluation invokes conservative parallelism. Speculative parallelism proves worthwhile if the arguments are eventually needed, but wastes effort if one or more of the arguments are never needed. Speculative evaluation of lengthy unneeded tasks could tie up processors, denying access to other higher priority tasks. Worse yet, speculative tasks could fail to terminate, removing their processors from further productive activity. Again, conservative parallelism simply postpones evaluation of speculative arguments until they are needed, perhaps bypassing some parallelism opportunities.

The eager evaluation of unneeded arguments is speculative parallelism at its worst where none of the speculation is productive. Lazy evaluation just ignores the unneeded arguments—it produces no parallelism but no effort is wasted either.

The inclusion of speculative parallelism significantly complicates the design of a parallel scheduler. Speculative schedulers usually require a priority scheduling scheme that is fair to equal priority tasks, and must be capable of canceling unneeded tasks. Conservative schedulers have less need to prioritize tasks and are never required to cancel tasks.

Implementation Alternatives The following summarizes the alternatives for conservative and speculative implementations. Assume that eager tasks are invoked in parallel.

- *Purely Eager Evaluation*—Has all of the benefits and problems associated with maximum speculation. Only ALICE uses this alternative and it is ineffective at controlling unnecessary speculative tasks.
- *Lazy Evaluation with Eager Annotations on Strict Tasks*—Implicitly annotated systems such as AMPGR, GRIP, HDG, and SISAL use this alternative. Tasks annotated by the compiler are sparked on available processors, but since they are strict tasks they are sure to be needed. Therefore, implicitly annotated systems implement purely conservative parallelism.
- *Lazy Evaluation with Eager Annotations on Some Strict and Non-Strict Tasks*—Explicitly annotated systems such as

Parallel SML, ParAlfl, <ν,G>, GAML, PAM, and PABC use this alternative. The annotated tasks are sparked on available processors. Other than ALICE, explicitly annotated systems are the only current implementations that implement speculation, and the programmer makes the speculation decisions.

Only <ν,G> and Qlisp attempt to address the problem of killing unneeded tasks created during speculation. <ν,G> uses its garbage collector to remove them from the system.

Qlisp also uses the garbage collector, but since the garbage collector is usually called upon infrequently, two other explicit task deletion methods are provided.

The first, kills explicitly named processes. For example:

`kill-process` *name*

kills the process identified by name.

The second method kills more general processes. Its behavior is modeled after the `catch` and `throw` primitives used in Common Lisp. Recall that Common Lisp tree searches can be terminated efficiently when the desired element is found by having the locating task invoke a `throw` to a `catch` in the initial search generation task. This causes termination notification to proceed from the central location of the root node.

`Throw` and `catch` have the same purpose in Qlisp except in Qlisp the tasks may be running in parallel. Therefore, `throw` and `catch` may initiate the termination of speculative tasks.

Each time a structure-oriented system divides the problem in two it speculates on those two components. HyperM parallelizes every task that results from the division and has no mechanism to kill unneeded speculative tasks.

Figure 25 summarizes how randomly parallel, explicitly annotated, implicitly annotated, and structure-oriented systems address conservative and speculative parallelism.

Compiler Speculation Compiler annotations applied by AMPGR, GRIP, HDG, and SISAL invoke the eager evaluation of strict tasks in parallel (e.g., tasks that are known to be needed at compile time). Therefore, they invoke purely conservative parallelism. Unfortunately, conservative implementations occasionally bypass parallel opportunities. On the other hand, speculative implementations sometimes waste processor time or unneeded tasks may fail to terminate. An integrated approach might exploit the advantages of one while avoiding the disadvantages of the other.

Explicitly annotated systems solve the problem by granting the programmer authority to speculate. This emboldens the programmer with power, but it also leaves speculation at the mercy of the programmer, a choice that may or not be wise.

If a programmer can do it, why can't a compiler do the speculating? Some feel that this would be a step in the wrong direction. They argue that the compiler is not sophisticated enough to make speculation decisions whereas humans are, or even if the compiler can be made sophisticated, the administrative costs would eat away any derived benefits. In any case, compile-time (and possibly even run-time) speculation is an alternative that should be investigated. Maybe it is a viable alternative in some cases.

In [Peyton Jones, 1989], Simon Peyton Jones makes the following observations about *integrated* (e.g., speculative and conservative) systems:

- Conservative tasks should run in preference to speculative tasks. For example in the expression if e_1 then e_2 else e_3, the conditional e_1 is conservative and should

Implementation	Conservative/ Speculative	Method used to eliminate unnecessary speculative tasks
Randomly Parallel	Speculative	None
	Conservative	N/A
Explicitly Annotated	Speculative	Qlisp uses either the garbage collector or explicit annotations to delete speculative tasks
		$<\nu,G>$ uses the garbage collector to delete speculative tasks
		In all other explicitly annotated systems, speculative tasks are not deleted at run time
Implicitly Annotated	Conservative	N/A
Structure-Oriented	Speculative	None

Figure 25: Conservative/Speculative Parallelism

have priority over the speculative tasks e_2 and e_3.

- Speculative tasks may become conservative. For example, when the conditional e_1 completes execution, one of e_2 and e_3 will become conservative, and the other will become unneeded (unless it is shared). The priority of a speculative task should be increased when it changes to a conservative task. Speculative tasks that become unneeded should be killed (along with their offspring).
- Speculative tasks should be scheduled fairly so that the speculation efficiently explores all alternatives.

Is speculation really wise? Well, sometimes it is and sometimes it is not. If a number of speculative tasks are sparked at the same time, and each of them in turn sparks a flood of subtasks, the system could become overloaded. The subtasks could exhaust a resource such as memory.

Even if there are sufficient resources to support a large number of tasks, what if some of the initial tasks are found to be unneeded? In the worst case, what if all of them are unneeded? Each must be killed along with their subtasks. In what order do you kill them? What if one or two are forgotten? The compiler must be reasonably sophisticated to handle situations like these. Furthermore, the killing process will likely consume huge amounts of processor time.

The killing of even a small number of speculative tasks is a problem. Some say a garbage collector could serve the dual role of removing garbage from the graph and killing unneeded speculative tasks [Wadler, 1987]. However, by the time the garbage collector responds, tasks may have consumed (wasted) many resources. For example, large chunks of processor time could be wasted.

A rule prohibiting the invocation of nonterminating speculative tasks is impossible to enforce. If it is possible to determine whether a speculative task will terminate, then it is possible to solve the halting problem, a classic unsolvable problem in computing theory. Maybe massively parallel systems are the answer. If non-termination were encountered in such a system it would merely incapacitate a few of the many processors. This incapacitation may not block or impede the program solution process.

The designers of current parallel implementations certainly considered compile-time speculation. The fact that none chose to include it in their design is testimony to the difficulty of the task.

4.3 Memory Organization

Memory organization is an important consideration when designing parallel functional systems. Although many memory designs are possible, the three alternatives considered in this paper are single address space or *shared-memory*, multiple disconnected address spaces or *distributed memory*, and a combination of the two called *hybrid memory*.

Shared-Memory *Shared-memory* is a single storage space available to a number of processors through a single addressing scheme. If memory access times for each of these processors is the same, the system is said to be a *uniform-access* shared-memory system; otherwise, it is a *non-uniform-access* system [Quinn, 1993].

GAML, <ν,G>, and Qlisp are implemented in uniform-access shared memory on the Sequent Balance, Sequent Symmetry, and Alliant FX/8 machines respectively, while AMPGR and Parallel SML use the BBN Butterfly's non-uniform-access shared-memory organization. ALICE and Flagship arrange the Transputer's distributed memory organization into a single non-uniform-access shared space.

In graph-based systems, shared-memory is a convenient place to locate the graph. Individual processors access graph components associated with their redex using simple machine operations and update the graph when reduction is complete. Therefore, as execution proceeds, shared-memory contains the current state of the graph. Referential transparency guarantees that processor operations will not interfere with one another.

Normally, a single processor queue is also located in shared-memory and all processors access it for tasking. This centralizes control and makes it possible to evenly balance processor workloads (i.e., provide *load balancing*). Unfortunately, only one processor can access shared queues at a time, so access must be coordinated by some type of shared locks. Contention for shared locks increases as the number of processors accessing them increases. This places a limit on the number of processors that can be added to shared-memory systems and still enjoy satisfactory performance gains.

Caching Locality of reference refers to the proximity of data and tasks that access that data. Good locality means most data references are local (i.e., on the same processor and in the same memory as the task), so data is referenced quickly. With bad locality, data must frequently be communicated from processors remote to the accessing task.

If each processor has local memory in addition to shared address space, one way to improve locality of reference is to store local copies of remote graph nodes in the sparking processor's local memory. This is called *data caching* [Peyton Jones, 1989].

These data caches are different from imperative language caches in the following ways:

- The most likely unit of caching is the graph component associated with a redex. Since redexes can vary in size this means the cache must accommodate variable rather than fixed length data.
- Unlike imperative cached data, functional graph components are usually not stored in contiguous cache locations. This complicates the transfer and fetching of cached data. Therefore, the latency of a cache miss may be substantial.

If graph components are shared, cache coherence can be a problem, but it is less of a problem in graph reduction systems than imperative systems. The first task to evaluate a graph node can mark it with an *ac-*

cessed annotation. Other tasks encountering that annotation will block until the first task completes its evaluation, updates the evaluated node with its result, and clears the blocking annotation. This means the cache should use a write-through policy so the annotation is entered in and removed from the global graph.

Distributed Memory In *distributed memory* systems, each processor has its own local memory. Non-local access is possible only by passing messages from one processor to another.

PAM, PABC, and HDG are implemented in distributed memory on the Transputer.

Since there is no shared-memory to store the program graph in, the graph is normally divided into components and distributed among the available processors. Then each processor locally reduces its assigned component.

Since each graph component could contain a number of redexes, each processor must have its own task scheduling queue. This is a more decentralized form of task control than that described above for shared-memory systems. The immediate benefit of decentralized control is that processors do not have to compete with each other for access to shared locks. The disadvantage is decentralized control requires explicit interprocessor communication to access non-local data, transfer results, and evenly distribute the processor workload during execution.

Referential transparency can reduce the level of communications. For example, if each processor receives an unshared graph component, there may be no need to synchronize or otherwise communicate with other processors until the graph component has been reduced to normal form (§ 1.4.2). Even then, the required communication only blocks links between the communicating processors. Other processor links remain free to handle other communication requirements in parallel.

Hybrid Memory Hybrid systems organize memory into both shared and distributed address spaces. The motivation for doing this is that if both organizations are available, the best features of each can be employed.

The following two alternatives can be used to implement hybrid memory systems.

1. Processor memories are organized into a single shared space, except part of each processor's memory is dedicated to a local cache that is not available to other processors. Collectively, the local caches form a distributed space. This offers the advantages of centralized addressing and control in the shared space and faster execution (if remote addressing is unnecessary) in the local processor memories.

   GRIP implements this alternative using MC68020 processors and SISAL implements it on the nCUBE2.

2. A number of local processor memories can be arranged into n memory clusters. Memory organization within each cluster is shared, but the organization between clusters is distributed. The advantage of this organization is it provides a good tasking hierarchy. Small tasks can be executed quickly in a single processor's local memory and medium-sized tasks can be executed in a cluster's shared-memory with no need for remote accesses. Only large-sized tasks overlap cluster boundaries and require remote access.

   HyperM implements this alternative using 4-processor memory MC88000 clusters.

Figure 26 summarizes memory organization alternatives used by current functional programming language implementations.

<table>
<tr><th colspan="2">Memory Organization</th><th>Implementation</th><th>Hardware</th></tr>
<tr><td colspan="2" rowspan="3">Uniform-access shared space</td><td><ν,G></td><td>Sequent Symmetry</td></tr>
<tr><td>GAML</td><td>Sequent Balance</td></tr>
<tr><td>Qlisp</td><td>Alliant FX/8</td></tr>
<tr><td colspan="2">Non-uniform-access shared space</td><td>AMPGR
Parallel SML</td><td>BBN Butterfly—
MC68020</td></tr>
<tr><td colspan="2">Distributed arranged into non-uniform-access shared space</td><td>ALICE
Flagship</td><td rowspan="2">Transputer</td></tr>
<tr><td colspan="2">Distributed space</td><td>PAM
PABC
HDG</td></tr>
<tr><td rowspan="3">Hybrid</td><td rowspan="2">Each processor has a local memory cache. Collectively, the processor caches form a distributed-memory. Processor memories outside the cache are connected into a single large shared-memory space.</td><td>GRIP</td><td>Up to 20 circuit boards each containing 4-MC68020 processors and one IMU</td></tr>
<tr><td>SISAL</td><td>nCUBE 2</td></tr>
<tr><td>Distributed among clusters, shared within clusters</td><td>HyperM</td><td>Several 4-processor MC88000 clusters</td></tr>
</table>

Figure 26: Memory Organization

4.4 Task Blocking and Resumption

In spite of referential transparency, there are still two occasions where the blocking and resumption of tasks is necessary [Peyton Jones, 1989]. First, in a distributed organization tasks normally block when they access data on a remote processor. When the data becomes available, the blocked task is resumed. Second, graph nodes are frequently shared. To avoid redundant evaluation, only one task should be allowed to reduce a shared component. Others should block upon attempting evaluation, to be resumed when the blocking process updates the graph structure.

Distributed-memory implementations like PAM, PABC, HDG and SISAL all block when remote nodes are accessed, and all parallel implementations address the redundant evaluation problem. Therefore, virtually all implementations resort to some form of task blocking and resumption. Normally, a field in the accessed expression's root node identifies that the expression is either shared or remote. Access to a remote node usually blocks the accessing task immediately, to be resumed when the required data overwrites the blocking node.

If the node is shared, the first accessing task is allowed to continue, but it invokes a locking mechanism that blocks further access to that node while evaluation is in progress [Peyton Jones, 1987; Peyton Jones, 1989; Field and Harrison, 1988]. Frequently, the blocked processes are added to a list originating at another field in the root node data structure. When evaluation completes and the root node is rewritten, blocked processes are reactivated (e.g., by adding the list of blocked processes to the active tasking pool).

The following are blocking and resumption alternatives:

Notification Model In the *notification model*, the parent blocks when a value is needed from a child. When the child has the value, it notifies the parent and the parent resumes execution. The parent can also block on a notification count and require notification from several children before unblocking. Blocking the parent may be unnecessary if the child has not begun to evaluate the desired task. For this reason, the notification model can be inefficient.

Evaluate and Die Model In the *evaluate-and-die model*, the parent attempts to evaluate the graph as though it never created a child. If the child has finished the evaluation, the parent simply accesses the updated value. If the child has not started the evaluation, the parent evaluates the graph as though the child was never created. In the latter case, if the child task is not shared it becomes an *orphan*. An advantage of evaluate-and-die is the parent blocks only if the child is currently evaluating the sub-graph. GAML and $<\nu,G>$ use the evaluate and die model.

Signal Set Notification ALICE and Flagship place reducible packets in an active pool. Selection of a function packet causes the packet to be rewritten if the function arguments are resolved, or blocked and added to a suspended pool if the arguments are unresolved. Blocked packets are in the *signal* set of the unresolved argument packets. Each time an argument is resolved, the signal set is used to notify the blocked packet. When all arguments are resolved, the blocked packet is transferred from the suspended to the active pool.

The ALICE and Flagship signal set mechanism is not designed to eliminate the redundant evaluation of shared nodes, but it can easily be adapted to do so. Multiple function packets can identify a particular packet as one of its arguments. All of the referencing packets are in the signal set of the argument. If the argument is not resolved, referencing packets will block. When the argument is resolved, all referencing packets (both blocked and unblocked) are notified.

Interrupt Notification Interrupts can be used to coordinate blocking and resumption. For example, SISAL implements blocking resumption as follows: When a remote reference is encountered, a request is sent to the processor containing the remote node and a new thread is added to the top of the requesting processor's activation stack. When the requested value returns, it will invoke an interrupt handling routine that adds the value and a presence bit to a *value array* that is indexed by frame number. Upon reactivation, the requesting thread checks the value array for the presence bit using its frame number as an index. If the presence bit is not set, the thread blocks again and another new thread is added to the activation stack. This process repeats itself until the requesting thread finds its presence bit set. It then accesses the value from the value array (again using its frame as an index) and continues execution.

Possibly Shared Nodes GAML reduces the scope of the evaluate-and-die problem by identifying nodes that can participate in sharing. It limits blocking and resumption consideration to only those nodes. Blocking and resumption overhead need not be applied to other nodes. The trade-off here is an increase in compile-time complexity for improved execution speed and simplicity.

Indirection Nodes HDG adds two new node types to the graph structure to facilitate sharing. The first is an input indirection node. It contains a reference count and points to a shared node in the evaluating processor's memory. With respect to blocking and resumption, it alerts evaluating process that the node is shared internally. Evaluation of the shared node applies the result to all referencing components.

The second is an output indirection node which points to a shared node on a re-

Blocking and Resumption Method	Implementation
Blocks on evaluation of function packets with unresolved parameters. Argument signal set reactivates blocked packets.	ALICE Flagship
Uses interrupts to implement both blocking and non-blocking communications. Remote references are coordinated by interrupts.	SISAL
Restricts blocking consideration to *possibly shared nodes* (determined at compile time).	GAML
Output indirection node points to shared external reference. Input indirection node contains reference count and points to shared internal node.	HDG
No blocking. Shared expressions are evaluated eagerly prior to distribution.	HyperM
Blocks for futures and critical sections. Both implicit and explicit resumption is provided.	Qlisp

Figure 27: Task Blocking and Resumption

mote processor's memory. This output indirection extends blocking and resumption to distributed-memory. It is HDG's main contribution to improving sharing. In contrast, implementations like PAM may re-evaluate remotely shared components.

Eager Evaluation Prior to Distribution HyperM evaluates shared components eagerly prior to sparking them on remote clusters. No work is duplicated because any duplication candidate is already reduced when the redex containing it is sparked. Therefore, blocking to prevent duplication caused by sharing is unnecessary. This simplifies execution issues at the expense of some loss of parallelism during eager evaluation.

Futures and Critical Sections In Qlisp, every new process has a data structure associated with it that identifies whether the process is evaluated or not. Processes that are not yet evaluated are called *futures*. Task bloking in Qlisp takes place either when one process requires the value of an incomplete future or when when a process wishes to access a busy critical section. Task resumption is either implicitly or explicitly invoked.

Implicit resumption takes place when futures are accessed. For example, if prop evaluates to `true` in:

$$\texttt{qlet}\ \textit{prop}\ \texttt{((x}_1\ \textit{arg}_1\texttt{)}\ldots\texttt{(x}_n\ \textit{arg}_n\texttt{)).}\textit{body}$$

any other process accessing *body* will block until the future (associated with *body*) is realized (i.e., arguments of the qlet are all evaluated).

Since `qlambda` expressions of the form:

$$\texttt{(qlambda}\ \textit{prop lambda-list.body}\texttt{)}$$

are atomic, only one process at a time can evaluate them. Others block and await their turn (resume implicitly).

Qlisp also provides for explicit resumption. For example;

$$\texttt{qwait(}\textit{future}\texttt{)}$$

causes a *future* to be invoked and blocks the calling process until the *future* is realized.

Qlisp also provides two type of *locks* to explicitly protect critical sections. The first, called a *spin* lock, causes blocked processes to loop repeatedly (busy-wait) awaiting entry to the critical section. The other, called a *sleep* lock removes blocked processes from the processor and places them in a waiting queue.

Figure 27 summarizes blocking and resumption alternatives unique to some current functional programming language implementations.

4.5 Task and Data Distribution

The even distribution of tasks among processors is an important issue in parallel design. To improve data access times, this distribution should also locate data in close proximity to the tasks that access that data. In this case the graph can be viewed as data (a resource) that is reduced by a task (executable instructions).

In uniform-access shared-memory systems where all processors have uniform access to memory, it makes little difference where tasking pools and data are located. However, in non-uniform access shared-memory and distributed-memory systems, the location of tasking pools and resources in different processor memories could significantly degrade performance.

Resource access times can be improved in distributed and non-uniform access shared-memory systems if the task scheduler and resource allocator communicate with one another when making processor assignments [Haines and Böhm, 1991]. For example, when a task is assigned to a particular processor, the data it will use should be located in that processor's memory or in the memory of a nearby processor. However, early data assignments may become less appealing as changes caused by the initiation of new tasks and task completions occur.

The same thing goes for the early assignment of tasks. In fact, it may become necessary to transfer tasks from one processor to another in order to balance the work load among processors. This is referred to as *load balancing* or *task migration*. Load balancing schemes are normally either *sender initiated* or *receiver initiated* [Haines and Böhm, 1991].

In sender initiated migration, the sender detects that the number of tasks in its scheduling queue awaiting execution is above a certain threshold and either offloads some of the tasks randomly to another processor (*blind migration*), or makes an intelligent decision regarding the processor to send the tasks to (*coordinated migration*).

In receiver initiated migration, the receiver detects that the number of tasks in its scheduling queue is below a certain threshold and either requests tasking directly from a random processor (blind migration) or makes an intelligent decision on which processor to receive tasking from (coordinated migration).

Diffusion migration represents a compromise between blind and coordinated migration. Transfers are coordinated, but less coordination is necessary because the transfer is only between a fixed set of nearby processors (e.g., with the north, east, south, and west neighbors in a mesh). The idea is to provide some transfer alternatives and yet keep transfer decisions simple and the tasks in close proximity to their data. Blind and coordinated migration allows tasks to jump anywhere in the processor structure.

If the resources are not moved along with the tasks that use them, access times can increase significantly (because communications distance between tasks and resources is increased). On the other hand, moving the resources with the tasks increases the complexity of load balancing and requires the transfer of more data during migration.

No matter how load balancing is implemented, tasks should only be allowed to migrate a fixed maximum number of times to prevent thrashing [Peyton Jones, 1989].

The remaining paragraphs in this subsection describe how current uniform-access shared-memory, non-uniform-access shared-memory, distributed-memory, and hybrid memory implementations address task and data distribution issues.

Uniform-Access Shared-Memory Systems The proximity of tasks and their data is not an issue in uniform-access shared-memory systems because the time to access data does not change with its placement in memory. Therefore, only task migration must be considered in shared systems.

$<\nu,\mathrm{G}>$—The $<\nu,\mathrm{G}>$ machine allocates and de-allocates space for the graph as large chunks of heap space located in shared-memory. When explicit sparking annotations are encountered during execution, the new tasks are added to a global tasking pool. Each processor has a local cache of sufficient size to store a local tasking pool and in the form of linked frames. Processors are tasked from the local pool. When a local pool is empty, the associated processor queries the global pool for work. The processors execute tasks until both the local and global pools are empty.

After initial distribution, tasks do not migrate from their assigned processors.

Access to the global pool is not protected so tasks are occasionally lost. This is not a problem because the evaluation transformer method requires parents to evaluate their children if they are not already evaluated.

The programmer controls the number of tasks and task grain size in $<\nu,\mathrm{G}>$ with explicit sparking annotations.

GAML—GAML's task and data distribution is similar to $<\nu,\mathrm{G}>$. Data is allocated and de-allocated in large hunks to fixed memory locations in the shared-memory heap. When sparked by explicit annotations, tasks are routed to the least busy processors, but no further migration is performed to balance a lopsided load. However, GAML limits the programmer's ability to control the number of tasks and task grain size by overriding explicit annotations whenever the system load is heavy.

Qlisp—Qlisp adds new tasks to a single queue located in shared-memory. Processors request work from the queue and then execute their assigned task to completion before requesting another. Neither data nor tasks migrate after their initial assignment. A single cache located in shared-memory is available to all processors.

Non-Uniform-Access Shared-Memory and Distributed Systems It is in non-uniform access shared-memory and distributed systems where coordination of task and data migration becomes important. It would be best if tasks were moved along with their data, but that alternative can be expensive. Diffusion migration is often used as an alternative where tasks are moved but not the data. Reasonable proximity is maintained because tasks can only migrate to neighboring processors, one hop away from data that is left behind.

AMPGR—A two-level scheduling strategy is used where tasks are initially assigned to the local tasking pools of available processors. Their execution may spawn other tasks that overfill the local pool. In that case, excess tasks are forwarded to a global tasking pool located in non-uniform access shared-memory where they are redistributed.

Data distribution is coordinated with the distribution of tasks. When a task is sparked at a processor, the graph reducer first tries to allocate space for the data in the processor's local memory. If no space is available, the data is allocated in the heap of a remote processor. Once space for data has been assigned, however, there is no facility to move it with tasks during load balancing.

AMPGR uses two mechanisms to control the number of tasks and to increase task grain size. First, one of a function's child tasks is executed on the sparking processor (i.e., at home). This reduces communications and ensures the sparking processor has work to perform while other processors are evaluating the other children. Also, the total number of system tasks is limited. When the limit is reached, processors evaluate tasks in-line rather than sparking them to other processors.

Parallel SML—Like AMPGR, a two-level scheduling strategy initially assigns tasks to the local tasking pools of available processors. Their execution may spawn other tasks that overfill the local pool. The excess tasks are forwarded to a global tasking pool located in non-uniform access shared-memory where they are redistributed.

ALICE and Flagship—In ALICE and Flagship, distributed memory space is organized into a non-uniform access shared space. Nodes of a program graph are represented as packets and these packets are placed into a pool in shared-memory. Processors select these packets randomly and attempt to modify them according to some packet rewrite rules. A function packet can be rewritten when all of its necessary arguments are available. Rewriting discontinues when the root node packet is rewritten with a value.

In this design, the packet function is the task and its arguments are the data. Before a function packet is rewritten, the required arguments must be transferred from global memory to the evaluating processor's local cache. The further away the arguments are, the more communication time is required.

In Flagship, load balancing information is communicated along with packets. When a packet is routed to a processor for evaluation, it is automatically routed to the least busy processor. Following routing, the load balancing information is readjusted. Unfortunately, this scheme does not take locality into consideration. Therefore, Flagship also attaches a *desired processor number* to each packet (e.g., processor nearest to memory containing packet arguments). However, when system load is heavy, the processor recommendation is ignored.

In Flagship, there is an active pool and a holding pool located in each processor's memory. Active pool tasks in excess of some predefined limit are reassigned to the holding pool where they are ignored until the load reduces to a level where they can be reactivated.

PAM—PAM derives parallelism from explicit user annotations and uses a distributed-memory organization. Each processor has its own graph reducer process and a separate communications process to distribute tasks. Initially, the program graph is distributed among the system processors. The graph reducer evaluates its graph component using the communications process to fetch remote arguments. The communications process is also responsible for task migration to effect load balancing. A simple diffusion migration strategy is used. An idle processor queries its neighbors for work. After any migration, the task is at most one hop away from its data.

The user explicitly controls the number of tasks and the task grain size.

PABC—Like PAM, PABC derives its parallelism from explicit user annotations and uses a distributed-memory organization. Furthermore, tasks and data are distributed as graph components at program start and each processor has a graph reducer process to reduce its graph component. PABC differs from PAM in that when each processor's communications unit transfers graph components to other processors, it is on a sub-component basis rather than a single node basis. Therefore, multiple data items can be fetched in a single communications request and data can be migrated with the associated tasks. Unfortunately, communications complexity is increased and the task of maintaining sharing is complicated.

The user explicitly controls the number of tasks and the task grain size.

HDG—Like PAM and PABC, HDG employs a distributed-memory organization, but it derives its parallelism from implicit compiler-derived annotations, not explicit annotations. Task distribution is regulated by two tasking pools per processor. The processor is assigned work from the *active* pool, while freshly sparked tasks are added to the *migratable* pool.

Tasks can migrate but data does not. When the processor needs work, it checks its own active pool first, its own migratable pool next, and the migratable pool of a direct neighbor last. This amounts to diffusion migration.

Hybrid Memory Systems Data position is unimportant in uniform-access shared-memory systems, but it is critical to good performance in non-uniform access shared and distributed-memory systems. On the other hand, contention for shared queues is a problem with shared-memory systems while there is no queue contention in distributed systems. In order to gain the benefits of both memory organizations, some designers blend them both into a hybrid memory structure. Some examples of this approach are described below.

SISAL—The nCUBE 2 SISAL implementation arranges one megabyte of each processor's memory into a single large shared-memory space. The remaining 'local' memory at each processor is not directly addressable by other processors. Collectively, the local memories form a distributed space.

SISAL compiles its source programs to 'C' code, which in turn is compiled to machine executable code. Therefore, there is no graph data structure at execution time. Instead, there is machine code and the data it accesses. To enhance performance, the code and scalar data structures are replicated in the local memory of all processors.

Vector data structures such as arrays, records, and streams are allocated in blocks of variable sizes and then are distributed equally across the shared-memory space. In an attempt to enhance locality, the size of blocks is variable, normally tied to the data size required by a loop.

Each processor has its own local ready queue. New tasks are added to the local queue whenever the currently executing task encounters a spawn or *fan out* instruction. Task activation records consist of pointers to the code and arguments for that task. Simple scalar parameters are passed by value while vector values are referenced where they are in shared-memory. Processes are blocked on remote reference, and are resumed by interrupt when the data becomes available.

The current SISAL implementation performs no task or data migration. Consequently, the load can become quite unbalanced and locality is a problem. A 'greedy' sender-initiated load balancing scheme is being considered by the designers.

GRIP—The GRIP design includes both shared- and distributed-memory. Groups of four-processor memories are connected together into an Intelligent Memory Unit (IMU), and IMUs are connected by a network into a single shared memory space. However, part of each processor's local memory is devoted to a cache which is not accessible by other processors. Collectively, these local memory caches form a distributed-memory space.

There are global tasking queues located at each IMU and local tasking queues located in each processor's cache. Processors receive direct tasking from the local queue.

The current configuration of the program graph resides in shared-memory. Each time a processor accesses a global node, a copy of the node and its substructure is is created in cache and is subsequently reduced there. Therefore, data is transferred to processors along with their tasks. Since the copying may duplicate work, the IMU sets a lock bit when a node is accessed the first time and adds subsequent accessing tasks to a waiting list. When the node is updated in global memory, the waiting tasks are added to the IMU tasking pool where they are distributed to local processors.

New nodes and the tasks associated with them are created in local cache, not in the IMU. Any time a local task completes, the entire reduced local sub graph is used to update the global graph. This *flushing* mechanism prevents remote pointers from appear-

<table>
<tr><th colspan="2">Placement of</th><th rowspan="2">New task placement</th><th rowspan="2">Tasked From</th><th rowspan="2">Tasking Remarks</th><th rowspan="2"></th></tr>
<tr><th>Graph</th><th>Tasking Pool(s)</th></tr>
<tr><td rowspan="3">Shared memory</td><td rowspan="3">Local & global</td><td>Local pool</td><td rowspan="3">Local pool</td><td>Spill-over to global pool—then redistributed</td><td>AMPGR
Parallel
SML</td></tr>
<tr><td rowspan="2">Global pool</td><td>Processor requests work</td><td><ν,G>
Qlisp</td></tr>
<tr><td rowspan="2">Distributed to least busy processor</td><td rowspan="2">GAML
ALICE</td></tr>
<tr><td rowspan="2">Shared memory/ locally cached packets</td><td rowspan="2">Global—
active & suspended
Local—
active & holding</td><td rowspan="2">Global active</td><td rowspan="2">Local active</td></tr>
<tr><td>Normal load—distributed to processor with best locality
Heavy load—distributed to least busy processor</td><td>Flagship</td></tr>
<tr><td rowspan="2">Distributed</td><td>Local</td><td>Local pool</td><td>Local pool</td><td>Diffusion scheduling</td><td>PABC
PAM</td></tr>
<tr><td>Local—
active & migratable</td><td>Local migratable</td><td>Local active</td><td>If local active pool is empty—consult local migratable pool first, then remote migratable pools</td><td>HDG</td></tr>
<tr><td rowspan="2">Shared memory/ locally cached redexes</td><td>Local & global</td><td rowspan="2">Local pool</td><td rowspan="2">Local pool</td><td>Tasks exported to global pool for redistribution when load is low</td><td>GRIP</td></tr>
<tr><td>Local</td><td>No Load Distribution</td><td>SISAL</td></tr>
<tr><td>Distributed/ shared within clusters</td><td>Cluster</td><td>Low complexity —local cluster
High complexity —remote clusters</td><td>Local cluster pool</td><td>Sandwich—divide-and-conquer</td><td>HyperM</td></tr>
</table>

Figure 28: Data Placement and Tasking

<table>
<tr><th colspan="2">Migration of</th><th rowspan="2"></th></tr>
<tr><th>Tasks</th><th>Data</th></tr>
<tr><td rowspan="2">Excess local tasks spill over into global pool and are redistributed</td><td>Attempt to allocate data space in local memory first, then in remote memory</td><td>AMPGR</td></tr>
<tr><td rowspan="4">No Migration</td><td>Parallel SML</td></tr>
<tr><td>Distributed from global pool; no further migration</td><td><ν,G>
GAML
ALICE
Flagship
Qlisp</td></tr>
<tr><td>Access neighbor migratable pools if no local tasks</td><td>HDG</td></tr>
<tr><td rowspan="2">Diffusion migration</td><td>PAM</td></tr>
<tr><td>Data migrates with tasks</td><td>PABC</td></tr>
<tr><td>Local tasks exported to global pool for redistribution when load is low</td><td>Redex copied from global pool to local cache when assigned to processor; redex copied from local cache to global pool when redistributed</td><td>GRIP
SISAL</td></tr>
<tr><td>Assigned to cluster; no further migration</td><td>Transmitted to remote cluster with task; no further migration</td><td>HyperM</td></tr>
</table>

Figure 29: Task and Data Migration

ing in the global graph.

When the system load is low enough, excess tasks are transferred from the local pool to the global IMU pool for redistribution and the sub graphs associated with those tasks are used to update the global graph. Consequently both tasks and data migrate at considerable communications expense.

HyperM—HDG memory space is shared within four-processor clusters, but each cluster forms a processing element in a distributed-memory space.

Each processor runs a divide-and-conquer reducer task called the *sandwich*. A disadvantage of this organization is that the program must either designed in or translated to divide-and-conquer form. Also, HyperM employs a hierarchical scheduling strategy that requires a complexity measure as input. The user provides this complexity measure by explicitly annotating the program tasks. For example, annotations for a sorting routine could range from very simple to complex based on the length of the list to be sorted.

Simple to moderately complex tasks are evaluated on the sparking processor's cluster where they can exploit shared-memory, while course grained tasks are distributed among two or more clusters. Since shared arguments are evaluated prior to distribution, both tasks and data can migrate to remote clusters with no fear of generating remote pointers.

Data placement and tasking information for current functional programming language implementations is summarized in figure 28. Task and data migration information is summarized in figure 29.

4.6 Task Grain Size

A large factor in the efficient operation of a functional programming language implementations is the average task size that processors must accommodate. As task grain size varies, so does the total number of tasks. An increase in grain size decreases the total tasks; a decrease in grain size increases the total tasks. Communication costs and context switching overhead imply that grain size should not be too small. Conversely, too large a grain size could reduce the tasks to a number less than the number of processors, failing to utilize the entire system.

Excessively large grain sizes are seldom a problem with functional programming language implementations. Normally the grain size is far too small and must be increased in size to be acceptable. The alternatives as grain size are:

- A single graph node—Which is far too small.
- A single redex—Which still is too small.
- A graph component that contains a number of redexes—Acceptable with the right number of redexes.

One of the following methods can be used to address grain size:

- Do nothing—The easiest alternative from an implementation standpoint, but costly in terms of efficiency.
- User annotations—Most explicitly annotated systems can increase task size by having the user annotate larger tasks.
- *In-line* tasks—Instead of sparking tasks to new processors evaluate them on the parent processor. This increases the number of redexes executed by the parent processor.
- *Cutoff* tasks—This is similar to in-lining except the tasks that would have been sparked are not scheduled for execution on the would-be sparking processor. The tasks could be evaluated in-line on the parent processor if the evaluate-and-die model is used. Otherwise, they are either lost or are placed in a holding queue to return and aggravate the grain size problem again later.

ALICE, Flagship, HDG, and SISAL do nothing to control grain size. SISAL notes that the small grain size hides remote latencies (an argument similar to the one used to support RISC hardware). ALICE and Flagship aggravate the grain size problem by scheduling tasks at the node level.

PAM, PABC, and $<\nu,G>$ use both user annotations and cutoff to control task size. GRIP also uses cutoff but GRIP and $<\nu,G>$ invoke inadvertent in-lining since the evaluate-and-die model causes tasks discarded by children to be eventually evaluated by the parent.

Qlisp also uses explicit annotations to control task size. Recall that *propositional parameters* are included in statements that invoke parallel tasks such as:

`spawn` *prop expression*

If the propositional parameter is `true`, the appropriate tasks (the task represented by expression in this case) are invoked in parallel with the current task. If the propositional parameter is `false`, the appropriate tasks are evaluated in-line with the current task. Consequently, the propositional parameter can be viewed as an explicit annotation to control task grain size.

HyperM is not classified as an explicitly annotated system, but it allows the user to apply complexity annotations that can be used to adjust the grain size.

On the other hand, GAML is an explicitly annotated system, but it is the compiler that invokes its parallel tasks based on system load. Tasks that are not sparked are evaluated in-line. AMPGR evaluates tasks in line by not sparking tasks that are already in head normal form, retaining the first child process on the parent processor, and evaluating tasks in-line when a certain threshold is reached.

Figure 30 summarizes the task grain size methods used by current functional programming language implementations.

Grain Size Control	Implementation
None	ALICE Flagship HDG SISAL
User Annotations and Cutoff	PAM PABC
User Annotations and Cutoff—Inadvertent In-Lining	<ν,G>
Cutoff—Inadvertent In-Lining	GRIP
Complexity Annotations	HyperM
In-Lining (By Compiler)	AMPGR GAML
In-Lining (By User Annotations)	Qlisp

Figure 30: Grain Size Control

4.7 Garbage Collection

Since most Functional Programming Language Implementations apply some form of graph reduction and the graph is most likely allocated in heap, garbage collection is an important consideration. The functional language implementations in this paper either do nothing about garbage or they apply one or a combination of the following techniques: reference counting, mark and sweep, and stop and copy. Each of these techniques is summarized below.

- *Do Nothing*—One way to deal with garbage is to simply ignore it. Any implementation that adopts this alternative does so in the interest of faster execution speed, sacrificing space, or maybe because it's the easiest alternative to design.

- *Reference Counts*—Reference Counting allocates a counter field to all nodes in the graph. Each structure's counter is initialized to zero and is incremented by one every time another structure references it and is decremented by one each time a reference is terminated. If the reference count is zero, the structure is garbage and can be reclaimed. Theoretically, if the count is greater than zero, the structure is referenced by some other structure and is therefore not garbage. However, this fails to consider cycles that may be disconnected from the program structure. Nodes involved in the cycles may be garbage but reference counting garbage collection would not reclaim them.

- *Mark and Sweep*—Mark and Sweep algorithms allocate a *marked/unmarked* field to all nodes in a graph. Initially all nodes are initialized to unmarked. Then all paths from the root node are followed changing the field of each structure encountered along the way to marked. When finished, all unmarked nodes are garbage. A major problem with mark and sweep is that it is extremely difficult to find the pointers to follow in variable sized nodes.

- *Stop and Copy*—Stop and copy algorithms normally compact the graph into one contiguous data area. The compaction requires time, but once complete garbage does not occur in the compacted

Garbage Collection Method(s)	Implementation
None	AMPGR
Reference Counting	ALICE PAM
Mark-and-Sweep	Qlisp
Reference Counting (Primary) Mark-and-Sweep to Reclaim Cycles	Flagship
Stop and Copy	GAML <ν,G> Parallel SML
Stop and Copy—Local Reference Counting—Global	HDG PABC
Stop and Copy—Local and Global	GRIP
Stop and Copy—Clusters	HyperM

Figure 31: Garbage Collection

structure. Another benefit is that allocation of new nodes simply amounts to advancing the free space pointer by the size of the new node. Stop and copy is the most widely used garbage collector due to its speed and ability to work with variable-sized nodes.

Only AMPGR chooses to ignore garbage collection. ALICE and PAM use reference counting, Qlisp uses Mark-and-Sweep, and GAML, <ν,G>, Parallel SML, and HyperM use stop and copy. HyperM has a hybrid memory organization so it applies the algorithm by task within its shared-memory clusters.

All of the other implementations use two garbage collectors to reclaim space. Flagship uses reference counting as its primary garbage collector but invokes mark and sweep occasionally to reclaim cycles. HDG and PABC both use stop and copy as a fast local memory garbage collector and reference counting as a global collector.

GRIP uses two stop and copy garbage collectors. Recall that GRIP's memory organization is hybrid. One compacts local memories quickly, while the other to compacts global memory at a slower pace.

Garbage collection methods used by current functional programming language implementations are summarized in figure 31.

4.8 Aggregate Memory Access

Most modern computer architectures are designed to access memory one word at a time. They are called *von Neumann architectures* in recognition of John von Neumann, one of the originators. The word-at-a-time limitation is frequently referred to as the *von Neumann bottleneck*, and languages designed for von Neumann architectures are called von Neumann languages [Backus, 1978].

Unfortunately, the von Neumann bottleneck prevents many current computers from accessing and manipulating multiple data items expeditiously (such as vectors and lists). This type of data is commonly referred to as *aggregate data*. In order to handle aggregate data efficiently, modifications to the von Neumann architecture are necessary (e.g., the addition of parallel memories and multiple processing elements). However, even when parallel hardware is available, language features may not allow programmers to exploit them. A good parallel language should include these aggregate features.

Combining Forms For that reason, John Backus includes combining forms such as *construction* and *apply-to-all* in his functional language FP [Backus, 1978]. Construction applies a list of functions to a single data object and apply-to-all applies a single function to aggregate data objects.

Backus chooses to limit FP to a small number of fixed combining forms arguing that this limited number can be applied in organized and manageable framework, whereas a greater number would only lead to chaos.

Although APL is not a functional language, it also uses combining forms. Unlike FP, APL's combining forms are numerous and very general. This enables APL to apply compact and succinct syntax when manipulating arrays and lists [Iverson, 1962].

Recognizing that the combining forms used by FP and APL are well suited for processing aggregate data, Budd and Pandey have proposed the following λ-based scheme to exploit them on parallel architectures [Budd and Pandey, 1991].

Since pure λ-calculus contains no simple aggregate data forms, they have extended the basic set of lambda definitions to include a *kappa-form* (used to construct data aggregates) and a *sigma-form* (used to reduce data aggregates).

Collection (κ) Form—The *collection* or κ (kappa) form uses labels to identify the composition and size of data aggregates. It is formed as follows:

$$\kappa \; size \; (\lambda \; x. \; expression)$$

The *size* field is used to store the data aggregate's size. It has the form $(dim, length)$. For example, $(0,1)$ is a structure with dim $(e.g., dimension) = 0$ and $len(e.g., length) = 1$, or a constant. *Size* $(1,200)$ identifies a one-dimensional array of length 200.

The κ-form's lambda expression determines the contents of the structure identified in the *size* field. For example:

$$\kappa \; (1,200) \; (\lambda p. \; p+1)$$

is a one-dimensional array of size 200 initialized with the numbers 0 to 199.

The κ-form is a single value (structure) that can be manipulated as an aggregate like the value of any other λ expression. When it is manipulated in conjunction with other aggregates, the κ-form imposes no specific ordering on the evaluation of its elements. This is important when generating parallel code because it allows flexibility.

For example, the outer product is a composite function meaning it consists of one function, $\circ$, that takes another dyadic scalar function, *op*, as its argument. Outer product performs *op* on all pairs of elements taken from the two arguments. For example, if *op* is $*$, outer product applied to vectors $\kappa \; (1, \; 4) \; (\lambda \; p.p) \; \equiv \; 0 \; 1 \; 2 \; 3$ and $\kappa \; (1, \; 4) \; (\lambda \; q.q+1) \; \equiv \; 1 \; 2 \; 3 \; 4$ can be expressed in Budd and Pandey's notation as:

$$(\kappa \; (1,4) \; (\lambda \; p.p)) \circ .* \; (\kappa \; (1,4) \; (\lambda \; q.q+1)) \equiv$$

$$(1 \; 2 \; 3 \; 4) \; \circ .* \; (0 \; 1 \; 2 \; 3) \equiv \begin{array}{rrrr} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \\ 1 & 4 & 16 & 64 \end{array}$$

The first row is $1^0 \; 1^1 \; 1^2 \; 1^3$ (the first element of 1 2 3 4 to the power of the elements in 0 1 2 3); the second row is $2^0 \; 2^1 \; 2^2 \; 2^3$ (the second element of 1 2 3 4 to the power of the elements in 0 1 2 3); and so on.

Translation of $(a \; \circ .op \; b)$ for this example is shown in figure 32.

The 10th element (i.e., element (2, 2)), would be computed as follows:

$$(a.item \; (10 \; \mathsf{div} \; 4)) * (b.item \; (10 \; \mathsf{mod} \; 4)) \equiv (a.item \; 2) * (b.item \; 2) \equiv 3 * 2 \equiv 9$$

Reduction (σ) Form The *reduction* or σ (sigma) form is used to reduce aggregate data types to a single scalar value. It consists of a

$$\lambda.\ a,\ b\ (\kappa\ ((a.dim + b.dim),\ (a.len,\ b.len))\ (\lambda\ p.\ (a.item\ (p\ div\ a.len))\ op\ (b.item\ (p\ mod\ b.len)))) \equiv$$

$$\lambda.\ a,\ b\ (\kappa\ ((2,\ (4,\ 4))\ (\lambda\ p.\ (a.item\ (p\ \mathsf{div}\ a.len)) *\ (b.item\ (p\ \mathsf{mod}\ b.len))))$$

Figure 32: Translation of $(a\ \circ .op\ b)$

function (e.g., addition) and a data generator as follows:

$$\sigma\ fun\ limit\ (\lambda\ x.\ expression)$$

During reduction, the λ-expression is repeatedly evaluated with arguments for the formal parameter (i.e., x above) ranging from 0 to lim. The function specified by the fun field is then applied to the set of values reducing them to a scalar value (e.g., the λ-expression results are summed).

4.9 Vectorization

Many languages such as FORTRAN-90 and APL allow programmers to apply standard mathematical operations such as addition and multiplication to aggregate data objects like arrays (instead of just applying them to scalar values). These aggregate objects are called vectors. For example, in FORTRAN-99, if A, B, and C are vectors, $A = B + C$ adds B and C element-by-element and stores each result in the corresponding location of array A. In APL, the same action is represented by $A \leftarrow B + C$. If the implementation is sequential, the vector operations can be performed one at a time using a process called *dragthrough* (e.g., start at the first element of the vector and continue (dragthrough) to the last index) [Budd, 1984]. If a compiler performs these operations concurrently, it is called a *vector compiler* and a vector compiler is said to perform *vectorization*.

5 Summary

Functional languages are very different from imperative languages. An important difference is that functional languages associate variable names with values (single assignment variables), whereas imperative languages associate variable names with memory locations (multiple assignment variables) (§ 1.2). Proponents of functional languages argue that in spite of any differences, functional languages are just as powerful as imperative languages and that the simplicity of single assignment variables can translate to better program readability and, perhaps, more reliable programs. Furthermore, executable functional-language tasks do not impose any side effects on each other. This characteristic, called *referential transparency*, can simplify translation, particularly in parallel implementations.

An objective of functional-language implementations is to realize these benefits, and in doing so to become more competitive with the imperative programming language paradigm. The progress of sequential and parallel functional-language implementations in this regard is summarized in the following paragraphs.

Sequential Implementations In order to be considered worthwhile, sequential implementations must support lazy evaluation and compilation with low operational overhead, but not all of them do.

Abstract Syntax Tree (AST) interpreters and the Object **S**tack, **E**nvironment, **C**ontrol, and **D**ump (SECD) machine do not handle either lazy evaluation or compilation efficiently (§ 2.1 and § 2.2). They are both environment-based techniques meaning that they store variable bindings in memory when the bindings are formed, and retrieve the bindings from memory when they are needed. The SECD machine outperforms AST interpreters because its stacks are a convenient place to store the environment. However, it is difficult if not impossible to reduce binding overhead to an acceptable level in environment-based systems.

Fixed-combinators do support lazy evaluation, can be compiled, and do not incur the same administrative overhead as environment-based systems (§ 2.3). However, they suffer from another damaging overhead problem. Since the fixed-combinators are small, task grain size is likewise small. This results in a penalty at execution time that degrades performance.

Graph reduction of lambda expressions lends itself well to lazy evaluation (§ 2.4), but because of free variables (§ 1.4.4), compilation is not straightforward. However, lambda lifting can remove the free variables. If lambda lifting is performed properly, the result is a maximally free expression called a super-combinator (§ 2.5). These super-combinators can be fully lazy and they can also be compiled reasonably efficiently (§ 2.6).

Compiled super-combinators compilers and source-to-source compilers (§ 2.6) seem to be emerging as the translation methods of choice as evidenced by the use of source-to-source in recent SISAL implementations and the use of both methods in different implementations of Haskell.

Optimizations In spite of their success at achieving compiled and fully lazy implementations, unoptimized functional implementations still do not achieve levels of performance comparable with imperative implementations. A number of optimization techniques (§ 3) narrow the gap.

Certain tail recursive functions delay some computations unnecessarily, resulting in a waste of time and space. Tail recursion elimination (§ 3.1), performed either by the programmer or by the compiler, can rearrange computations and reduce the delays. This technique can conserve space, speed up execution, or both.

Although lazy evaluation requires the sharing of common components inside an expression during evaluation, it fails to consider sharing in some cases, such as calling a function with the same arguments as an earlier call. Memoization corrects this problem by establishing a cache for previously computed results (§ 3.2).

Imperative memoization uses an imperative language compiler to perform memoization, whereas applicative memoization uses a single functional-language compiler to to perform both translation and memoization. Imperative memoization tends to be faster because it allows the use of iterative looping.

Sometimes function arguments are unnecessarily copied during the reduction of λ-expressions. It may be possible to identify these arguments in advance and alter evaluation to avoid the copying. Techniques such as λ-Argument Copying (§ 3.3) do eliminate the copying but at the expense of heavy administrative overhead.

Partial Evaluation assigns values to some arguments of a multiple-argument function and partially evaluates the function body with respect to those values at or before compile time (§ 3.4). If the values are correct (i.e., they turn out to be the ones that are needed), the partially evaluated function will require less work to reduce during execution than the original function.

Finally, although lazy evaluation is very expressive, eager evaluation is frequently the more efficient alternative of the two. Consequently, some translators apply either lazy or eager evaluation to program expressions depending on the circumstances. Such translators are said to apply mixed-order evaluation (§ 3.5).

Generally, mixed-order translators evaluate all program expressions lazily except those specially marked (e.g., those annotated by the user or compiler) for eager evaluation. The annotated expressions are normally strict (i.e., their result is definitely required).

It can be argued these optimization techniques and the compilers and interpreters they enhance do not exhibit the elegance of design that imperative implementations do. Perhaps that is because more attention has been devoted to imperative languages than functional languages, or perhaps it is because we are more used to abstracting away imperative language complexities. In any case, a goal of functional implementation designers should be to narrow this gap and/or counter the perceived notion that functional-language implementations are not as elegant as imperative implementations.

Parallel Implementations Since the introduction of parallelism to functional-language implementations is recent, the long term implications of parallel implementation features are quite unclear.

Parallel functional-language implementations use four methods to invoke parallelism: random, explicitly annotated, implicitly annotated, and structure-oriented (§ 4.1). In random implementations, each processor draws work components blindly from a pool of contenders and executes those components as tasks. In explicitly annotated systems, users insert special annotations into source code. During execution the annotated tasks are run in parallel. Parallel activity is also invoked by annotations in implicitly annotated systems, but it is the compiler, not the user, that inserts implicit annotations into code. In explicitly annotated systems, the programmer controls parallel activity whereas in implicitly annotated systems parallelism is transparent to the programmer. Finally, structure-oriented systems require that programs either be in, or transformed to, a certain structure. For example, the structure could be one that implements a divide-and-conquer algorithm. Then tasks identified at each divide-and-conquer step would be invoked in parallel.

Parallel tasks are invoked with either conservative or speculative parallelism (§ 4.2). In conservative parallelism, only needed tasks are invoked in parallel. Strict functions and arguments are always needed, so conservative parallelism always invokes them in parallel. Non-strict tasks may or may not be needed so conservative implementations do not invoke them in parallel. On the other hand, speculative implementations take a chance on some non-strict tasks by invoking them in parallel. The chance proves worthwhile if the task is later determined to be needed, or it leads to wasted effort if the task is found to be unneeded.

Structure-oriented and explicitly annotated systems must invoke speculative parallelism, whereas all current implicitly annotated systems choose to invoke conservative parallelism. Random implementations are mainly speculative but can achieve some awkward measure of conservative parallelism.

Only $<\nu,G>$ and Qlisp address the problem of killing speculative tasks when they are found to be unneeded. Both use their garbage collector to remove tasks and residue from the system. However, since the garbage collector is invoked infrequently, Qlisp provides two additional explicit task deletion alternatives. One kills tasks in conjunction with `catch` and `throw` primitives that are used to terminate operations (such as a search when the desired item is found). The other kills specifically named processes (i.e., `kill-process` *name*).

No single memory organization is preferred by parallel functional language implementors (§ 4.3). Actual implementations are fairly evenly divided between uniform-access shared-memory, non-uniform-access shared-memory, distributed memory, and hybrid memory (i.e., combinations of shared and distributed memory). Many implementations assign each processor a cache to speed up local operations.

There also is little correlation between task blocking and resumption methods employed by parallel systems (§ 4.4). Each implementation that blocks uses a method that is quite different from the others and which is based on its own requirements.

It is disappointing that functional implementations need to consider task blocking and resumption at all. Due to referential transparency (§ 1.2) executable functional-language tasks are free of side effects and therefore cannot interfere with one another.

However, blocking is necessary for other reasons. For example, in non-uniform-access shared-memory systems (where a single address space is divided among a number of processors) if a task on one processor accesses an address located on another processor, the accessing task may block until the requested component has been supplied. Also, to avoid duplication of effort, processors accessing a shared expression that is already being evaluated by another processor may block and wait until the computation is complete and the result is available.

There appears to be no single plan used to handle data and task placement (§ 4.5). A number of combinations of local and/or global tasking queue organizations are implemented where control is either centralized or localized and data accessed by tasks may or may not be located near the processor executing those tasks.

It seems appropriate that steps be taken to balance the load during execution by migrating tasks (and data required by those tasks) from busy to lightly-loaded processors. However, many implementations in this survey perform no migration at all. Those that do most often use a very simple method such as diffusion migration (migration only between neighboring processors). Furthermore, most migrate only to balance tasks, making no attempt to move data associated with the tasks. Of course this migration simplicity is adopted to avoid the overhead required to support more complex migration. However, the simplicity may incur penalties in other areas such as poor performance due to load imbalance or due to a large number of remote accesses.

As is the case with migration, parallel implementations tend to opt for simplicity when dealing with the size of tasks (i.e., task grain size) (§ 4.6). Many make no attempt to adjust task grain size. Since grain size is usually too small, this can lead to a large number of parallel tasks, and the administration required to start up and support these parallel tasks can eat up performance gains. Other implementations use a cutoff to limit the number of tasks. When a certain threshold of task activity is reached, subsequent tasks are suppressed (not invoked) until the activity dies down. Only a few implementations retain selected parallel tasks on the invoking processor (rather than spawning them on other processors). This technique called *in-lining* does increase task grain size.

There is no uniform agreement on the type of garbage collector to be used with parallel functional-language implementations (§ 4.7). The alternatives represented in this survey are none (i.e., no garbage collection), reference counting, mark-and-sweep, stop and copy, and various combinations of the above. The more aggressive implementations have separate local and global garbage collectors sometimes with different methods applied at each level.

None of the parallel functional-language implementations mention aggregate memory access or seem to place much emphasis on vectorization. It could be that these features are included but discussions about are suppressed for some reason. However, it is more likely that they are either not included or are not exploited to their full potential. If so, they are practical performance improvement features that deserve consideration in future implementations.

Analysis Based on the contents of this survey, the following comments on functional-language implementations are offered for consideration:

The best alternative to perform sequential operations appears to be compiled supercombinators. It is necessary to resort to this form of compilation rather than imperative compilation techniques because functional languages frequently pass functions as parameters and include call-by-need parameter passing (whereas most imperative languages do not include these language features).

Each of the sequential optimization techniques (§ 3) seem to be considered in isolation. Some attention ought to be directed towards integrating the techniques together, and determining how they interact. Furthermore, there is virtually no mention of these optimization techniques in any of the parallel implementation papers included in this survey. Certainly many of the techniques are just as applicable to parallel systems.

Unfortunately, referential transparency does not appear to be realizing some of its expected benefits. For example, recall that because of an absence of side effects, referentially transparent expressions do not interfere with one another during execution. This should translate to less blocking in parallel functional implementations than in parallel imperative implementations, but, in fact, the survey indicates that a good deal of blocking is still necessary. True, referential transparency is not the cause of the blocking, but the fact that blocking is still needed is disappointing.

Conservative parallelism bypasses some parallel opportunities, so to optimize parallelism it is sometimes necessary to speculate. Explicitly annotated systems speculate wisely only if the programmer speculates wisely. If that is not the case, the execution of unnecessary tasks can wipe away the advantages of parallel activity.

The feasibility of an implicitly annotated system that speculates should be investigated. A simple heuristic could used to identify suitable speculative tasks (fallible, of course, due to the halting problem). Once invoked, the system would need to monitor speculative tasks removing them and their residue when they are determined to be unneeded. Such a system would be expected to outperform conservatively parallel systems, but it would probably not achieve the performance of elegant programs designed for explicitly annotated systems.

Migration in parallel implementations needs a lot of work to determine what to migrate (i.e., data and/or tasks), where to migrate it (i.e., to a neighbor or to any processor), and when to migrate. Current migration policies are too simplistic showing only pockets of creativity and little regard for the penalties of inaction.

The average size of tasks, or task grain size, is a problem in parallel implementations. In-lining parallel tasks (retaining them on the invoking processor for execution) can correct the problem but only if prudent decisions are made on which tasks to retain. As is the case with speculation above, such decisions are bound to be fallible because of the halting problem (the execution time of tasks is unknown at the time the tasks are scheduled).

Most sequential and parallel implementations can capitalize on the benefits of aggregate memory access and vectorization, but few of them seem to do so. It is possible that these features were de-emphasized in early implementations in order to reduce overall complexity. They should included in the basic design of future systems.

Finally, a very important matter that seems to be largely neglected in parallel implementations is portability. Most of the current systems are implemented in fairly specific hardware that is not widely available elsewhere. This impedes improvement since only selected sites can exercise the system. The creators of Haskell understand this problem and are directing their efforts towards a system that sits on top of Parallel Virtual Machine (PVM) [Sunderam, 1992] and runs on a network of workstations.

A Introduction to Haskell

The September 1987 Conference on Function Programming Languages and Computer Architecture in Portland, Oregon determined that there were more than a dozen non-strict, purely functional programming languages, all with similar semantics and expressive power. A consensus of the conference attendees felt widespread use of functional languages was hampered by a lack of a common language. A committee was formed to design a new language, one with strong semantic power and highly expressive syntax. The result of this effort is *Haskell*, a programming language named in honor of the logician Haskell B. Curry who contributed to the development of the Combinatory Logic (§ 2.3) [Curry and Feys, 1958].

A rigorous description of the Haskell syntax and semantics is contained in [Hudak and Wadler, 1988] while a more limited description intended only to introduce the language is contained in [Hudak, 1989].

A brief introduction to Haskell is presented here for two reasons. First, because it is expressive and this clarity makes it a good representative of the functional programming language paradigm. Second, all functional program segments in this survey are presented in Haskell (except for those that describe another language's syntax and semantics). The introduction is very limited, intended only to provide an appreciation of Haskell's expressiveness while at the same time making it possible to interpret the survey's program segments.

Formal Parameters Haskell does not enclose formal parameters in parentheses, nor does it separate formal parameters with commas. The mathematical function:

$$\texttt{f}\ (\texttt{x}_1,\ \texttt{x}_2,\ \texttt{x}_3,\ \ldots\ ,\texttt{x}_n) \;=\; \texttt{expression}$$

for $n > 0$ would be represented in Haskell as:

$$\texttt{f}\ \texttt{x}_1\ \texttt{x}_2\ \texttt{x}_3\ \ldots\ \texttt{x}_n = \texttt{expression}$$

Pattern Matching Haskell applies pattern matching to its formal parameters. The general structure of a Haskell program is:

$$\begin{array}{lcl} \texttt{f pat}_1 & = & \texttt{exp}_1 \\ \texttt{f pat}_2 & = & \texttt{exp}_2 \\ \texttt{f pat}_3 & = & \texttt{exp}_3 \\ \vdots & & \vdots \\ \texttt{f pat}_n & = & \texttt{exp}_n \end{array}$$

where `f` is a function and $\texttt{exp}_i$ is the expression or function body for the function applied to parameter pattern $\texttt{pat}_i$ for $1 \leq i \leq n$.

Consider the following example adapted from [Hudak, 1989]. It is a program to determine whether an element `x` is a member of a list:

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

One way to represent lists is by placing a list of elements inside brackets. Therefore, $\texttt{elt}_1,\ \texttt{elt}_2,\ \texttt{elt}_3,\ \ldots, \texttt{elt}_k$ is a list of k elements and `[ ]` is the empty list. The first pattern of `member` is `x [ ]` where the parameter `x` is the element to be matched and the parameter `[ ]` is the empty list. If this pattern is matched, the function returns `False`.

The structure `(elt:sub-list)` can also be used to represent lists where `:` is similar to the LISP `CONS` operator. Parentheses enclose the list for clarity. The second pattern of `member` is `x (x:xs)` where again `x` is the element to be matched and `(x:xs)` is a non-empty list.

The third pattern of `member` is `x (y:xs)`. If pattern matching proceeds from top to bottom, the pattern where $\texttt{y} = \texttt{x}$ must have failed when matching of the previous pattern was attempted. Therefore, $\texttt{y} \neq \texttt{x}$. The expression invoked by this pattern recursively calls upon `member` with `x` and the reduced list as parameters.

Qualifying Parameters It is possible to limit the values of Haskell parameters or limit values of elements within parameters by placing a | operator and a relational expression after the element. Haskell relational operators are ==, /=, >=,<=, <, and >. If the third pattern of member were changed to:

```
member x (y|x/=y:xs) = (member x xs)
```

the patterns of member would reduce to two cases with the empty list matched first.

Lambda Expressions It is convenient to use λ-expressions such as λ x.+ x x (§ 1.4) in functional programs. Unfortunately, the λ character is not an ASCII character. Haskell solves this problem by replacing λ with the backwards slash character. Additionally, it replaces the period following the formal parameter with an arrow and uses infix rather than prefix operators in the body of the λ-expression. For example:

```
λx.+ x x
```

would appear in Haskell as:

```
\x -> x + x
```

Arrays The following notation is used to define arrays in Haskell:

```
array(low,up)[(i,expr)|i<-[min..max]]
```

The keyword array identifies the expression as an array function. The first parameter provides the lower (low) and upper (up) bounds of the array using the syntax (low,up). The second parameter is read "index i has value expr where i is between the values of min and max." For example,

```
array(1,10)[(i,i*i)|i<-[1..10]]
```

is an array where the array value at index i is i*i for 1 $\leq$ i $\leq$ 10.

The exclamation point is used to identify elements of an array. For example,

```
array(1,10)[(i,i*i)|i<-[1..10]] ! 5
```

identifies the array's fifth index and has the value 5*5.

User-Defined Types Haskell allows the user to define data types. For example, the user defines or instantiates (§ 1.4.2) the variable a as an array type with the statement:

```
data a =
      array(1,10)[(i,i*i)|i<-[1..10]]
```

Within the scope of this definition a ! 7 would return the value 7*7 or 49.

Acknowledgments

Our sincere appreciation is extended to those who contributed to this survey. Special recognition is extended to César Quiroz, Margaret Burnett, Roy Crosbie, and Paul Luker who served as valuable consultants. Many thanks.

References

[Abelson *et al.*, 1985] Harold Abelson, Gerald Sussman, and Julie Sussman, *Structure and Interpretation of Computer Programs*, McGraw Hill, 1985.

[Aho *et al.*, 1986] Alfred Aho, Jeffrey Ullman, and Ravi Sethi, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[Appleby, 1991] Doris Appleby, *Programming Languages: Paradigm and Practice*, McGraw Hill, 1991.

[Augustsson, 1984] L. Augustsson, "A Compiler for Lazy ML," In *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, pages 218–227, Aug 1984.

[Augustsson and Johnsson, 1989]
L. Augustsson and T. Johnsson, "Parallel Graph Reduction with the $<\nu$,G$>$-Machine," In *Fourth Conference on Functional Languages and Computer Architecture*, pages 203–213, 1989.

[Böhm *et al.*, 1991] A. P. Böhm, D. C. Cann, J. T Feo, and R. R. Oldehoeft, "SISAL 2.0 Reference Manual," Technical Report CS-91-118, Colorado State University, Nov 1991.

[Backus, 1978] John Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," *Communications of the ACM*, 21(8):613–641, 1978.

[Baily, 1985] R. Baily, "FP/M Abstract Syntax Description," Internal Report, 1985.

[Banach *et al.*, 1988] R. Banach, M. Greenberg, J. Sargeant, I. Watson, P. Watson, and V. Woods, "Flagship: A Parallel Architecture for Declarative Programming," In *Fifteenth IEEE/ACM Symposium on Computer Architecture*, pages 124–130, 1988.

[Barendregt *et al.*, 1992] H. P. Barendregt, M. Beemster, P. H. Hartel, R. Hofman, K. G. Langendoen, L. L. Li, R. Milikowski, J. C. Mulder, and W. G. Vree, "Programming Clustered Reduction Machines," Technical Report CS-92-05, University of Amsterdam, 1992.

[Budd, 1984] Timothy Budd, "An APL Compiler for a Vector Processor," *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, Jul 1984.

[Budd and Pandey, 1991] Timothy A. Budd and Rajeev K. Pandey, "Compiling APL for Parallel and Vector Execution," In *Proceedings of the International Conference on APL*. Stanford University, 1991.

[Burn, 1991] G. L. Burn, *Lazy Functional Languages: Abstract Interpretation and Compilation*, Pitman Publishing, London, UK, 1991.

[Burstall *et al.*, 1980] R. Burstall, D. MacQueen, and D. Sanella, "Hope: an Experimental Applicative Language," Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, 1980.

[Burton, 1984] F. W. Burton, "Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs," *TOPLAS*, 6:151–174, 1984.

[Church, 1941] Alonzo Church, *The Calculi of Lambda Conversion*, Princeton University Press, 1941.

[Clack and Peyton Jones, 1985a] C. D. Clack and Simon Peyton Jones, "Generating Parallelism from Strictness Analysis," Internal Note 1679, February 1985.

[Clack and Peyton Jones, 1985b] C. D. Clack and Simon Peyton Jones, "Strictness Analysis—A Practical Approach," In *Conference on Functional Programming and Computer Architecture, Nancy*, 1985.

[Curry and Feys, 1958] Haskell Curry and Robert Feys, *Combinatory Logic*, volume 1, North Holland, 1958.

[Darlington and Reeve, 1981] J. Darlington and M. Reeve, "A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages," In *Proceedings of the ACM Symposium on Functional Languages and Computer Architecture*, pages 65–76, Oct 1981.

[Davie, 1992] Antony Davie, *An Introduction to Functional Programming Systems Using Haskell*, Cambridge University Press, 1992.

[Eisenbach, 1987] Susan Eisenbach, *Functional Programming, Languages Tools, and Architectures*, Ellis Horwood, 1987.

[Field and Harrison, 1988] Anthony J. Field and Peter G. Harrison, *Functional Programming*, Addison Wesley, 1988.

[Field, 1990] John Field, "On Laziness and Optimality in Lambda Interpreters: Tools for Specification and Analysis," In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

[Fisher and Leblanc, 1988] Charles Fisher and Richard Leblanc, *Crafting a Compiler*, Benjamin Commings, 1988.

[Friedman and Wise, 1976] D. P. Friedman and D. S. Wise, *Automata, Languages, and Programming*, Edinburgh University Press, 1976.

[George, 1989] L. George, "An Abstract Machine for Parallel Graph Reduction," In *Fourth Conference on Functional Programming Languages and Computer Architecture*, pages 214–229, 1989.

[George and Lindstrom, 1992] L. George and G. Lindstrom, "Using a Functional Language and Graph Reduction to Program Multiprocessor Machines or Functional Control of Imperative Programs," In *IEEE*, 1992.

[Goldman and Gabriel, 1988] R. Goldman and R. P Gabriel, "Qlisp: Experience and New Directions," *ACM Sigplan Notices*, 23(9):111–123, 1988.

[Goldman and Gabriel, 1989] R. Goldman and R. P. Gabriel, "Qlisp: Parallel Processing in Lisp," *IEEE Software*, pages 51–59, July 1989.

[Gordon *et al.*, 1979] M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF*, Springer-Verlag LNCS 78, 1979.

[Haines and Böhm, 1991] Matt Haines and Wim Böhm, "Towards a Distributed Memory Implementation of Sisal," Technical Report CS-91-123, Colorado State University, Nov 1991.

[Hall and Wise, 1987] C. Hall and D. Wise, "Compiling Strictness into Streams," In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 132–143, 1987.

[Hammond and Peyton Jones, 1991] K. Hammond and Simon Peyton Jones, "Profiling Strategies on the GRIP Parallel Reducer," Technical Report Internal Report 10, University of Glasgow, Scotland, 1991.

[Henderson, 1980] P. Henderson, *Functional Programming Applications and Implementation*, Prentice Hall, 1980.

[Holst and Gomard, 1991] Carsten K. Holst and Carsten K Gomard, "Partial Evaluation is Fuller Laziness," In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 223–233, Sep 1991.

[Hudak, 1986a] Paul Hudak, "Para-Functional Programming," *IEEE Computer*, pages 60–70, 1986.

[Hudak, 1986b] Paul Hudak, "Para-Functional Programming: A Paradigm for Programming Multiprocessor Systems," In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 243–254, 1986.

[Hudak, 1989] Paul Hudak, "Conception, Evolution, and Application of Functional Programming Languages," *ACM Computing Surveys*, 21(3):359–411, Sept 1989.

[Hudak and Wadler, 1988] Paul Hudak and Philip Wadler, "Report on the Functional Programming Language Haskell," Tech. Rep. YALEU/DCS/RR-656, Yale University, 1988.

[Hughes, 1984] R. J. M. Hughes, *The Design and Implementation of Programming Languages*, PhD thesis, Oxford, Sept 1984.

[Hughes, 1990] R. J. M. Hughes, "Compile-Time Analysis of Functional Programs," *Research Topics in Functional Programming*, pages 117–153, 1990.

[Iverson, 1962] Kenneth Iverson, *A Programming Language*, Wiley, 1962.

[Johnsson, 1984] T. Johnsson, "Efficient Compilation of Lazy Evaluation," In *Proceedings of the ACM Conference on Compiler Construction*, pages 58–69, Jun 1984.

[Keller and Sleep, 1986] Robert Keller and M. Ronan Sleep, "Applicative Caching," *ACM Transactions on Programming Languages and Systems*, 8(1):88–108, Jan 1986.

[Kingdon *et al.*, 1991] H. Kingdon, D.R. Lester, and G. L. Burn, "The HDG-Machine: a Highly Distributed Graph-Reducer for a Transputer Network," *The Computer Journal*, 34(4):290–301, 1991.

[Kuo and Mishra, 1987] Tsung-Min Kuo and Prateek Mishra, "On Strictness and its Analysis," In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 144–155, 1987.

[Lamping, 1990] John Lamping, "An Algorithm for Optimal Lambda Calculus Reduction," In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 16–30, 1990.

[Landin, 1964] P. J. Landin, "The Mechanical Evaluation of Expressions," *Computer Journal*, 6:308–320, 1964.

[Langendoen, 1993] Koen Langendoen, *Graph Reduction on Shared-Memory Multiprocessors*, Febodruk, Enschede, Holland, 1993.

[Loogen *et al.*, 1989] R. Loogen, H. Kuchen, K. Indermark, and W. Damm, "Distributed Implementation ofl Programmed Graph Reduction," *Parallel Architectures and Languages Europe (PARLE), LNCS 365/366*, pages 136–157, 1989.

[Maranget, 1991] L. Maranget, "GAML: A Parallel Implementation of Lazy ML," In *Fifth Conference on Functional Programming Languages and Computer Architecture LNCS 523*, pages 102–123, 1991.

[McCarthy *et al.*, 1962] J. McCarthy, P. Abrahams, D. Edwards, T. Hart, and M. Levin, *LISP 1.5 Programmers Manual*, MIT Press, 1962.

[Michie, 1968] D. Michie, ""Memo" Functions and Machine Learning," *Nature*, pages 19–22, 1968.

[Mycroft, 1981] Alan Mycroft, "The Theory and Practice of Transforming Call-by-Need into Call-by-Value," *LNCS 83, Springer Verlag*, pages 269–281, 1981.

[Nöcker *et al.*, 1991] E. Nöcker, M. Plasmeijer, and S. Smetsers, "The Parallel ABC Machine," *Implementation of Functional Languages on Parallel Architectures*, pages 351–382, 1991.

[Nielson, 1987] Flemming Nielson, "Strictness Analysis and Denotional Abstract Interpretation," In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 120–131, 1987.

[Peyton Jones, 1987] Simon Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall International, 1987.

[Peyton Jones, 1989] Simon Peyton Jones, "Parallel Implementations of Functional Programming Lanugages," *The Computer Journal*, 32(2):175–186, 1989.

[Peyton Jones *et al.*, 1987] Simon Peyton Jones, C. Clack, J. Salkild, and M. Hardie, "GRIP–A High Performance Architecture for Parallel Graph Reduction," *Third Conference on Functional Programming Languages and Computer Architecture, LNCS 274*, pages 98–112, 1987.

[Peyton Jones and Lester, 1992] Simon Peyton Jones and D. Lester, *Implementing Functional Languages: A Tutorial*, Prentice Hall International, 1992.

[Plasmeijer and van Eekelen, 1993] M. J. Plasmeijer and M. van Eekelen, *Functional Programming and Parallel Graph Rewriting*, Addison Wesley, 1993.

[Quinn, 1993] Michael J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw Hill, 1993.

[Schönfinkel, 1924] M. Schönfinkel, "Über die Bausteine der mathematischen Logik," *Mathematische Annalen*, 92:305–316, 1924.

[Sebesta, 1989] Robert Sebesta, *Concepts of Programming Languages*, Benjamin Commings, 1989.

[S.K.Skedzielewski and Glauert, 1985] S.K.Skedzielewski and J Glauert, *IF1—An Intermediate Form for Applicative Languages*, Lawrence Livermore National Laboratory, Livermore, CA, 1985, Manual M-170.

[Steel and Sussman, 1975] Guy Steel and Gerald Sussman, *Scheme: An Interpreter for the Extended Lambda Calculus*, MIT Artificial Intelligence Laboratory, 1975, Memo 349.

[Sunderam, 1992] Vaidy Sunderam, "Concurrent Computing with PVM," In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, dec 1992. Supercomputing Computations Research Institute, Florida State University.

[Takano, 1991] Akihiko Takano, "Generalized Partial Computation for a Lazy Functional Language," In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 1–11, Sep 1991.

[Wadler, 1987] Philip Wadler, "Fixing a Space Leak with a Garbage Collector," *Software Practice and Experience*, 17:595–608, 1987.

[Wadler, 1988] Philip Wadler, "Strictness Analysis Aids Time Analysis," In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 119–132, 1988.

[Wadsworth, 1971] C. P. Wadsworth, *Semantics and Pragmatics of the Lambda Calculus*, PhD thesis, Oxford, 1971.

[Watson and Watson, 1986] I. Watson and P. Watson, "Graph Reduction in a Parallel Virtual Memory Environment," In *Graph Reduction, LNCS 279*, pages 265–214. Springer-Verlag, 1986.

[Watson and Watson, 1987] I. Watson and P. Watson, "Evaluation of Functional Programs on the Flagship Machine," In *Third Conference on Functional Programming Language and Computer Architecture*, pages 432–434. Springer-Verlag, 1987.