Lisp and Symbolic Computation, 1,147–164 (1988)
© 1988 Kluwer Academic Publishers—Manufactured in The Netherlands

Code Optimizations for Lazy Evaluation

ADRIENNE BLOSS, PAUL HUDAK and JONATHAN YOUNG
Department of Computer Science, Yale University, New Haven, CT 06520

Abstract

Implementations of lazy evaluation for nonstrict functional languages usually involve the notion of a delayed representation of the value of an expression, which we call a *thunk*. We present several techniques for implementing thunks and formalize a class of optimizations that reduce both the space and time overhead of these techniques. The optimizations depend on a compile-time inferencing strategy called *path analysis*, a generalization of strictness analysis that uncovers order-of-evaluation information. Although the techniques in this paper are focused on the compilation of a nonstrict functional language for a conventional architecture, they are directly applicable to most of the virtual machines commonly used for implementing such languages. The same techniques also apply to other forms of delayed evaluation such as *futures* and *promises*.

1. Introduction

One of the hallmarks of modern functional languages is their *nonstrict* semantics, whose manifestation in an implementation is usually called *lazy evaluation*. Much recent work has been aimed at the efficient implementation of nonstrict functional languages. Many different approaches have been proposed, including the lazy SECD machine [9], G-Machine [14, 17], combinator machines [23], Categorical Abstract Machine (CAM) [20], Three-Instruction-Machine (TIM) [6], and others. Although the word "machine" is part of all these names, in fact they are all *virtual* machines that are typically implemented on conventional hardware. Thus, despite the variety of implementations, at the lowest level they all deal with the same small set of underlying issues.

One such issue is implementing lazy evaluation, or, more specifically, the need to *delay the evaluation of an expression until its value is absolutely needed.* In applicative-order evaluation ("call-by-value"), a function evaluates its arguments before it begins execution; thus, each argument is evaluated exactly once. In normal-order evaluation ("call-by-name"), an argument is evaluated each time its value is demanded inside the function; thus each argument is evaluated zero or more times. In lazy evaluation ("call-by-need"), an argument is evaluated when its value is first demanded, at which point the value is stored and returned to further demands; thus each argument is evaluated at most once. Normal-order is more powerful than applicative-order evaluation in that it may terminate in some cases where applicative-order evaluation does not. (If both strategies terminate, they are guaranteed to give the same result.) In the

This research was supported in part by the National Science Foundation under Grant DCR-8451415.

absence of side effects, lazy evaluation is semantically equivalent to normal-order evaluation, but it is often more efficient since it requires no recomputation. Thus, lazy evaluation is the choice of most modern functional languages, and its efficient implementation is of primary importance.

We refer to the run-time representation of a delayed expression as a *thunk.*[1] Each of the implementation strategies mentioned earlier manifests thunks in a different way, but in the abstract they all do the same thing: delay evaluation of an expression until its value is demanded, then evaluate it and save the value so that it may be used on future demands without recomputation. While this "delayed evaluation" is critical to the semantics of functional languages, creating and maintaining thunks contributes substantially to the overhead of executing functional programs. Strictness analysis [21] is aimed at reducing this overhead by detecting when it is safe not to delay an expression, that is, to convert from lazy evaluation (or call-by-need) into call-by-value. Although strictness analysis has recently been extended to handle higher-order functions [4,12], streams [7], and other nonflat constructs [13,24], it does not take us far enough: in our experiments only about half of all lambda abstractions are found to be strict [25]. The remaining functions still require delayed arguments, and so the need to implement thunks efficiently persists.

Our contribution in this paper is the design of several techniques for implementing thunks, and the formalization of a class of optimizations that reduce both the space and time overhead of these techniques. We have implemented most of the optimizations in our implementation of Alfl [10], a lazy functional language designed at Yale. The optimizations depend on a compile-time inferencing strategy called *path analysis* that uncovers order-of-evaluation information in a nonstrict functional language. While our models of thunks and their optimizations are expressed in Scheme code (which is the output of our compiler) to be run on a conventional uniprocessor, they are also directly applicable to the virtual machines mentioned earlier. Furthermore, when based on a modified form of path analysis, the same optimizations also apply to delayed evaluation in other contexts, such as *futures* in MultiLisp [8] and *promises* in Argus [19].

2. Motivation

2.1. The costs of lazy evaluation

Every implementation technique for lazy languages entails some overhead in delaying the evaluation of an expression.[2] In graph reduction the overhead is expressed largely in the creation and destruction of pieces of graph—thunks correspond to unevaluated

[1] The original use of the term *thunk* was in the context of the Algol implementation of call-by-name.
[2] Although some models of evaluation (such as Turner's strategy for combinator reduction [23]) appear to get lazy evaluation "for free," that is only because it is the *default* evaluation strategy; all of the virtual machines mentioned earlier can be optimized for strict evaluation, at which point the free lunch can be seen as an illusion.

parts of the graph, and evaluated values are "boxed" or tagged to distinguish them. In contrast, on a (virtual) stack machine, lazy evaluation is effected by creating a *closure*—some code along with bindings for its free variables. These closures must usually be allocated in the heap, and their evaluation requires a procedure call. In fact, at this level of the implementation, stack machine implementations look surprisingly similar to graph reduction, and similar descriptions apply to the CAM and the G-Machine.

Regardless of the implementation technique, the space and time overhead for a thunk can be summarized as follows:

1. Space complexity:
   - (a) Space for the *environment* (i.e., bindings for free variables).
   - (b) A cell to hold the computed value for later use.
   - (c) Tag bits to indicate the status of the evaluation.
2. Time complexity:
   - (a) The time required to create the thunk.
   - (b) The time required to invoke ("force") the thunk.
   - (c) The time required to update the thunk with the computed value.
   - (d) The time required to test the status of a thunk.

While the time to create a thunk (2a) is largely accounted for by space considerations, we must also account for the overhead of accessing the thunk's value (2b–d). And although the cost associated with something like testing the status of a thunk may seem trivial, such a test embedded within a tight inner loop can add a significant cost to the overall computation.

Although these costs are always present, their relative magnitudes can vary considerably from architecture to architecture. For example, on conventional architectures, the actual invocation of a thunk (2b) typically requires the saving and restoring of registers and other processor state (i.e., a context switch); however, the virtual machines mentioned above have very little state. Similarly, on most machines the status test (2d) requires a separate instruction, while some custom architectures can detect in hardware when access to an unevaluated thunk takes place (and trap to an exception routine). In this paper we will for the most part ignore these differences, but they may be important in optimizing thunks for a particular architecture, since some design decisions involve trade-offs between the various kinds of overhead.

2.2. Opportunities for optimization

As noted in the introduction, even after strictness analysis we cannot always convert from call-by-need into call-by-value. Fortunately, in these cases compile-time information about the *order in which expressions are evaluated* can often be used to manipulate thunks more efficiently at run time. For example, if we can determine that a certain demand for a value will definitely require computation of that value, we can eliminate the test for the status of the thunk (item 2d in the last section). If we can determine that a computed value will never be required again, it need not be stored (items 1b and 2c). In this paper we describe an entire spectrum of such optimizations that improve both time and space performance by using the order-of-evaluation information provided by path analysis.

Notation. Our source language is a notational variant of the untyped lambda calculus with constants. Application is written by juxtaposition ($f\,x$); abstractions are written $\lambda x.e$ or are named ($f\,x = e$). The conditional is written $p \rightarrow c,a$, and pairs are written $x{:}y$. We present concrete representations of thunks using Scheme [22] code.[3] To distinguish between source code and target code in the text, we write source code in italics, as in $(f\,x) + (g\,y)$, and target (Scheme) code in typewriter font, as in `(+ (f x) (g y))`.

3. Path analysis

Path semantics is a nonstandard semantics that describes *order of evaluation of expressions* for a lazy sequential functional language. *Path analysis* is an abstract interpretation of path semantics that provides compile-time information about order of evaluation. Path analysis subsumes standard strictness analysis, and the extra information that it provides allows a variety of optimizations beyond the standard conversion of call-by-name into call-by-value. The full theory of path semantics and abstract interpretation is beyond the scope of this paper. For more detail on the former, see [2, 3], and for the latter, see [1, 5]. In this section we give an overview of path analysis with an emphasis on the information required for optimizing thunks.

3.1. Intuitive description

A *path* through a function f is a totally ordered subset of f's formal parameters, where the ordering represents the evaluation order of those parameters in the body of f. We write $\langle x_1, x_2, \ldots, x_n\rangle$, $n \geqslant 0$, for a path with n elements where the x_i are formal parameters. Thus, a particular call to a function on a sequential machine has exactly one path, but at compile-time we can only infer a *set* of possible paths. Consider the following functions:

$$g\;a\;b \;\;= a + b$$

$$f\;x\;y\;z = (\mathrm{x} = 0) \rightarrow y, g\;x\;z$$

[3] In fact, our implementation is based upon a compiler for Scheme which implements closures (and in general any higher-order function) and known procedure calls very efficiently on stock hardware [18].

Assuming that $+$ evaluates its arguments left to right,[4] the set of paths through g contains only one element, $\langle a,b\rangle$. The set of possible paths through f is $\{\langle x,y\rangle, \langle x,z\rangle\}$, with the first and second paths corresponding to the consequent and alternate of the conditional, respectively. Note that although x is demanded twice if the alternate is taken, it appears only once in the second path. This reflects the "one-time evaluation" property of lazy evaluation: the first demand to x will cause it to be evaluated, but the second demand simply returns a stored value, and so does not contribute to the path.

3.2. Path semantics

We introduce D, the set of all variables, and *Path*, the flat domain of paths with bottom element $\perp_p$. *Path* is defined as follows:

$$Path = \{\perp_p\} \cup \{\langle d_1, \ldots, d_n\rangle \mid n \geqslant 0, \forall i, 1 \leqslant i < n, d_i \in \mathrm{D}\}$$

Note that the empty path $\langle\rangle$ is simply the path in which no bound variables are evaluated, and is not to be confused with the bottom path $\perp_p$, which represents nontermination.

We define "::" as an infix path append operator as follows: $\forall p \in Path$, $x_i \in \mathrm{D}$, $1 \leqslant i \leqslant n, n > 0$:

$$p :: \perp_p = \perp_p$$

$$\perp_p :: p = \perp_p$$

$$p :: \langle\rangle = p$$

$$\langle\rangle :: p = p$$

$$\begin{aligned}\langle x_1, \ldots, x_m\rangle :: \langle x_{m+1}, \ldots, x_n\rangle = \ & \text{if} \quad x_{m+1} \in \{x_1, \ldots, x_m\} \\ & \text{then } \langle x_1, \ldots, x_m\rangle :: \langle x_{m+2}, \ldots, x_n\rangle \\ & \text{else } \ \langle x_1, \ldots, x_m, x_{m+1}\rangle :: \langle x_{m+2}, \ldots, x_n\rangle\end{aligned}$$

Note that all but the first occurrence of a given bound variable are removed from a path; as discussed earlier, this is because a path reflects the order of *evaluation* of bound variables, and in lazy evaluation a bound variable is evaluated at most once.

An exact path semantics can be given which finds, for each function in a program, the path through that function for any given argument tuple. However, this semantics

[4] This assumption is to simplify this discussion only; as discussed in [2,3], the order in which a strict operator evaluates its arguments may be determined in any one of a number of ways, or may be left undetermined until run time.

is of no use to us since it is not effective; i.e., it is not guaranteed to terminate, since in general it requires knowing run-time values that depend on the standard semantics. What we need instead is an *approximation* to path semantics that is computable at compile time, and that still provides useful information. Such an approximation, called *path analysis*, is developed in [2,3] as an abstract interpretation of an exact path semantics. Instead of relying on the standard semantics, path analysis assumes, for example, that either arm of a conditional could be taken; it therefore finds a *set of possible paths* through a function, with the safety criterion that the actual path must appear in that set. Abstract interpretation is a natural tool for this sort of analysis because it readily yields an interprocedural analysis and is easily proven correct.

3.3. *Path analysis*

Semantic domains

$Path$,	the domain of paths
$P(Path)$	the powerdomain of *Path*
$Pfun = \cup_{n=1}^{\infty}(P(Path^n) \rightarrow P(Path))$	the function space mapping paths to paths
$Aenv = Fv \rightarrow Pfun$	the function environment
$Bve = Bv \rightarrow Path$	the bound variable environment

Semantic functions

$$\begin{aligned}
&\mathscr{A}: Exp \rightarrow Bve \rightarrow Aenv \rightarrow P(Path)\\
&\mathscr{A}_k: Pf \rightarrow Pfun\\
&\mathscr{A}_{pr}: Prog \rightarrow Aenv\\
\mathscr{A}_k[\![+]\!] &= \lambda s.\{x :: y, y :: x \mid (x,y) \in s\}\\
\mathscr{A}_k[\![IF]\!] &= \lambda s.\{p :: c, p :: a \mid (p,c,a) \in s\}\\
\mathscr{A}[\![c]\!]\, bve\ aenv &= \{\langle\rangle\}\\
\mathscr{A}[\![x]\!]\, bve\ aenv &= \{bve[\![x]\!]\}\\
\mathscr{A}[\![p(e_1,\ldots,e_n)]\!] bve\ aenv &= \mathscr{A}_k[\![p]\!](\mathscr{A}[\![e_1]\!] bve\ aenv \times \ldots \times \mathscr{A}[\![e_n]\!] bve\ aenv)\\
\mathscr{A}[\![f(e_1,\ldots,e_n)]\!] bve\ aenv &= aenv[\![f]\!](\mathscr{A}[\![e_1]\!] bve\ aenv \times \ldots \times \mathscr{A}[\![e_n]\!] bve\ aenv)\\
\mathscr{A}_{pr}[\![\{f_i(x_1,\ldots,x_n) = e_i\}]\!] &= aenv \text{ whererec}\\
aenv &= [(\lambda s. \cup \{\mathscr{A}[\![e_i]\!]\ [y_j/x_j]\ aenv \mid (y_1,\ldots,y_n) \in s\})/f_i]
\end{aligned}$$

Since we are interested in *sets* of paths, the basic domain is *P(Path)*, the powerdomain of *Path*.[5] The function space *Pfun* maps sets of tuples of paths (corresponding to tuples of arguments) to sets of paths, and the function environment *Aenv* takes a function variable and returns an element in the function space. The bound variable environment takes a bound variable and returns the path that it is currently bound to.

[5] For technical reasons that are not important here, we use the Egli-Milner powerdomain.

The meaning of the semantic equations is now straghtforward: The only path through a constant is the empty path. The only path through a bound variable is its path in the bound variable environment. The paths through the application of a primitive operator depend on its definition in K_a applied to the "cross product" of the sets of paths associated with the arguments.[6] The paths through a function call depend on the meaning of the function in the function environment applied to the cross product of the paths of the arguments. Finally, the "meaning" of a program is a function environment which, when given a function name and a set of tuples of paths, returns the set of paths through that function.

Perhaps the most insight into what path analysis actually computes comes from examining the primitive operations. "+" takes a set of tuples of paths, and for each tuple (x,y) returns the paths formed by appending x to y and by appending y to x; this reflects that the arguments to "+" could be evaluated in either order. The conditional also takes a set of tuples, and for each tuple (p,c,a) returns the paths formed by appending c to p and by appending a to p; this reflects that either arm of the conditional, c or a, could be taken after the predicate p is evaluated.

We have not yet given an algorithm for computing paths, but it follows directly from the above analysis by combining it with a technique for finding fixpoints on *Pfun*. We use the standard technique of "pumping with bottom" (with some caching optimizations), where the initial approximation to each function is $\{\perp_{Pfun}\}$, the bottom element in *Pfun*, and we iterate through the equations finding stronger approximations until we arrive at a fixpoint.[7] Our domains are finite and our functions on them are monotonic, so we are guaranteed to reach a fixpoint in a finite number of steps.

From a complexity standpoint, path analysis is at least as expensive as strictness analysis (which it subsumes), and strictness analysis has been shown to be exponential in the number of arguments to a function in the worst case [12]; in fact, path analysis is factorial in the number of arguments in the worst case. Fortunately, as is true for many such analyses (including strictness analysis), worst-case situations rarely occur in practice, and our experiments show that the number of iterations required is typically small. Furthermore, in our experience with functional programs, the average number of arguments to functions does not grow with the size of a program, and thus if an arbitrary bound is placed on the number of arguments, then the analysis can be shown to be linear in program size.

3.4. Using paths for optimizing thunks

As discussed in Section 2, many thunk optimizations rely on information about the *evaluation status* of expressions, in particular bound variables. For example, we want to know which demand for a bound variable forces its evaluation, and which

[6] This cross product produces sets of tuples from a tuple of sets, and results in a "relational attribute method" which is more accurate than the corresponding "independent attribute method" [15].

[7] For a description of efficient fixpoint finding techniques on finite spaces, see [26].

demand(s) simply return a previously computed value. However, a path as defined above contains at most one occurrence of any bound variable, and so does not contain information about the order in which the *occurrences* of a *particular* bound variable are demanded. Fortunately, once we have determined the path behavior for functions, we can infer the possible paths through occurrences as well; we accomplish this by deriving an auxiliary function in which the individual occurrences are distinguished by renaming. Thus for f defined in Section 3.1, we derive a new function f':

$$f'\ x_1\ x_2\ y\ z = (x_1\ = 0)\ \rightarrow\ y, g\ x_2\ z$$

f' is exactly like f except that it has two formal parameters for x, one for each of the occurrences, which are now named uniquely. Note that the set of possible paths through f' is $\{\langle x_1, y\rangle, \langle x_1, x_2, z\rangle\}$. We can now see that:

1. x is evaluated in *every* path through f, and thus f is strict in x (this is exactly the information provided by standard strictness analysis).
2. When x_1 is demanded, it is always the *first* occurrence of x in the path.
3. When x_2 is demanded, it is *never* the first occurrence of x.

Thus, x_1 will always force evaluation of the thunk associated with x, while x_2 will always return the thunk's stored value. In the next section we will see how this and other path information is useful in optimizing lazy evaluation.

4. Representations of thunks

Only two basic constructs are required to implement lazy evaluation: **DELAY**, which takes an expression and "wraps it up" into a thunk for later evaluation, and **FORCE**, which determines the value of a thunk. However, the way in which **DELAY** and **FORCE** are implemented can greatly affect their potential for optimization. We now present four different methods, or *modes*, for delaying and accessing values—closure mode (CL), cell mode (C), optimized cell mode (CO), and value mode (V)—and discuss various optimizations for each. We distinguish between different implementations of **DELAY** and **FORCE** by subscripting by the mode, e.g., $\mathbf{DELAY}_{CL}$ or $\mathbf{FORCE}_{CO}$.

4.1. The closure mode

The most obvious way to delay the evaluation of an expression is to enclose it within a parameterless procedure:

(DELAY exp) $\Rightarrow$ **(lambda () exp)**

in which case forcing a thunk is simply a function call:

(FORCE exp) ⇒ (exp)

Of course, this doesn't "cache" the computed value, as required by lazy evaluation. To store the value, and return it upon subsequent demands, we simply do the appropriate check and storage operations within the procedure:

```
(DELAY_CL exp)  ⇒  (let ((done '#F)
                          (val nil))
                     (lambda ()
                        (cond (done val)
                              (else (set! val exp)
                                    (set! done '#T)
                                    val))))
```

The **FORCE** operation is unchanged:

(FORCE$_{CL}$ exp) ⇒ (exp)

The subscript CL emphasizes that this is a *closure*. (This representation of a thunk might aptly be called a "self-modifying thunk.")

This mode may be optimized in special cases. Suppose that *exp* is either a constant or an expression that we know will be evaluated at most once; then caching the value is superfluous, since in the former case there is no evaluation to be done, and in the latter case there is no need to retain the computed value. Thus, in both cases we can simplify the implementation to

(TRIVIAL-DELAY$_{CL}$ exp) ⇒ (lambda () exp)

A less obvious optimization involves the "passing along" of bound variables. Consider the following function:

$$f\,x\,y = y \;\rightarrow\; 0,\, x + (g\,x)$$

In general f will create a thunk for the argument it passes to g. But that argument is just x, which was passed delayed to f, and so must be forced to get its value. So the call to g is implemented as

((FORCE$_{CL}$ g)(DELAY$_{CL}$ (FORCE$_{CL}$ x)))

However, it seems redundant to delay the force of an already delayed object, so we perform the following optimization:

(DELAY$_{CL}$ (FORCE$_{CL}$ x)) ⇒ x

Thus we have, **((FORCE$_{CL}$ g) x)** for the call to *g*.

Despite these optimizations, closure mode has two significant disadvantages. First, every strict reference to a bound variable *x* requires an *unknown procedure call*, that is, a call to the parameterless procedure representing the thunk bound to *x*. This is true even if *x* has already been evaluated and we are just returning a stored value. This call becomes expensive when variables are referenced often, since even very efficient procedure call mechanisms do not reduce the cost of context switching.

The second disadvantage is that most of the order-of-evaluation optimizations cannot be used. The reason is that the mechanism that decides whether to evaluate the expression or return the stored value is in **DELAY**, but the information used for the optimizations is available only when the expression is forced. Although occasionally we may be able to perform optimizations when we *create* the thunk (e.g., the **TRIVIAL-DELAY** optimization), a thunk is typically forced in several different places, and we would like to optimize each place separately. The next mode allows such optimizations.

4.2. *The cell mode*

In this mode a thunk is not represented as a function but as a pair, whose first element is a Boolean flag indicating the status of the second element: If true, the second element contains a value; if false, a parameterless procedure that will return the value when called:

```
(DELAY_C exp)  ⇒  (cons '#F (lambda () exp))
(FORCE_C exp)  ⇒  (if (car exp)
                       (cdr exp)
                       (let ((v ((cdr exp)) ))
                         (set! (cdr exp) v)
                         (set! (car exp) '#T)
                         v))
```

Using this mode we can perform optimizations analogous to those for the closure mode:

(TRIVIAL-DELAY$_C$ exp) ⇒ **(CONS '#T exp)** (1)

(DELAY$_C$ (FORCE$_C$ exp)) ⇒ **exp**

Note, however, that **(TRIVIAL-DELAY$_C$ exp)** does not delay evaluation of **exp** (as opposed to **(TRIVIAL-DELAY$_{CL}$ exp)**, which delayed **exp** but did not cache its value once evaluated), and so may be used with constant expressions but not with expressions that are evaluated at most once.

But now the disadvantages of the closure mode have gone away: Accessing a variable no longer requires an *unknown* function call (**IF, CAR,** and **CDR** are known functions) if the variable has already been evaluated, and the work of status-checking and caching the computed value is now done in **FORCE**.

To see how we can take advantage of order-of-evaluation information, suppose an occurrence of a bound variable x is *never* the first to be demanded, that is, we know that x has been evaluated before this demand. Then when we force it we no longer need to check whether it has been evaluated, and can simply return its value directly[8]:

$$\textbf{(FORCE}_C\ \textbf{exp)} \ \Rightarrow\ \textbf{(cdr exp)}$$

What if we know that a given occurrence of x is *always the first* to be demanded; that is, it will always force evaluation of the delayed expression? It seems that the following optimization is possible:

```
(FORCE_C exp)  ⇒  (let ((v ((cdr exp)) ))          (2)
                    (set! (cdr exp) v)
                    (set! (car exp) '#T)
                    v))
```

Unfortunately, this optimization is not always safe when combined with the other optimizations discussed above (1). To see why, consider the function f defined earlier:

$$f\ x\ y = y \ \rightarrow\ 0,\ x + (g\ x)$$

Suppose that + evaluates its arguments left to right—first x is evaluated, then (because we optimize **(DELAY (FORCE x))** ⇒ **x)** g is called with a thunk that has already been evaluated, that is, whose car is '#T. But if g uses the new optimization (2) for the first access of its argument, then it will try to "call" the cdr of the thunk, which is a value! The interaction arises because g assumes that its arguments are unevaluated, but because of the optimizations at (1) this may not be the case. Of course, we could do a global analysis to determine which functions always get their arguments unevaluated, and use this optimization in those functions only; this possibility should be weighed against (or possibly combined with) the next mode.

4.3. The optimized cell mode

This mode is just like cell mode except that a function's arguments are guaranteed to be unevaluated on entry to that function. That is,

$$\textbf{DELAY}_{CO}\ = \textbf{DELAY}_C$$

[8] Simple (local) versions of this optimization have been used in other compilers; for example, the Lazy ML compiler [14].

FORCE$_{CO}$ = **FORCE**$_{C}$

The difference is that both of the optimizations described for the first two modes are disallowed:

(DELAY$_{CO}$ **(FORCE**$_{CO}$ **exp))** $\not\Rightarrow$ **exp**

(TRIVIAL-DELAY$_{CO}$ **exp)** $\not\Rightarrow$ **(DELAY**$_{CO}$ **exp)**

However, the opportunities for using path information increase greatly, and may outweigh the loss of these two optimizations.

There are three basic pieces of information that are useful in optimizing **FORCE** for a particular occurrence of a variable x. This information, which is provided by path analysis, can be described as determining for certain that x:

- Has previously been evaluated (i.e., is this never the first occurrence of x?).
- Has never been evaluated (i.e., is this always the first occurrence of x?).
- Will never be evaluated again (i.e., is this always the last occurrence of x?).

If one of the first two conditions holds the status check may be eliminated, and if the third condition holds the computed value need not be cached. The following table shows how **FORCE**$_{CO}$ may be optimized under the resulting six combinations of compile-time data:

	Evaluated?	Unevaluated?	Unknown
Last?	x has already been evaluated: **(cdr x)**	This is the only demand for x: **((cdr x))**	x will never be demanded again: **(if (car x)** **(cdr x)** **((cdr x)))**
Unknown	x has already been evaluated: **(cdr x)**	x has not yet been evaluated: **(let ((v ((cdr x))))** **(set! (cdr x) v)** **(set! (car x) '#T)** **v)**	No information, no optimization possible.

4.4. *The value mode*

It is most effcient to access a value directly, that is, without delaying or forcing it. While this differs from the above modes in that it is not really a method for delaying computation, for expository purposes it is useful to view it as a mode in which

DELAY$_V$ and **FORCE**$_V$ are no-ops. We call this the *value mode*; it is the mode used by arguments which it is safe to pass by value, as determined by strictness analysis.

5. Code generation

5.1. Basic issues

Given the order-of-evaluation information required to perform the optimizations described above, the main issue in code generation is which mode to assign to each expression. Furthermore, because of the presence of higher-order functions in modern functional languages, determining the mode is not enough—we must also assign to each expression a "higher-order mode" that describes the mode to be used when passing an argument to the expression (function) as well as the mode the function will use when returning a value. We call these higher-order modes *interfaces*, and give them the following syntax:

Modes $m \in M = \mathrm{V} \mid C \mid CO \mid CL$

Interfaces $i \in I = m \mid m/(i_1 \Rightarrow i_2)$

A mode is one of V (value), C (cell), CO (cell optimized), or CL (closure), as discussed above. An interface i specifies the mode (m) to be used in accessing (forcing) a value, and if the interface is associated with a function value, we must also specify what interface (i_1) the function expects of its arguments, as well as the interface (i_2) that we can expect the value returned from the function to use.

Thus, the best (most efficient) interface for an expression denoting a strict function would be $V/(V \Rightarrow V)$—the expression is evaluated, arguments are passed in evaluated, and the result is evaluated. A lazy function, on the other hand, might be assigned the interface $V/(C \Rightarrow V)$, specifying that its arguments are passed in using the cell interface.

There are several issues involved in choosing the mode and interface for each expression. It is easy to show that choosing an optimal mode is undecidable, so we must use heuristics. The value mode is the most effcient, but is not always safe. When arguments must be delayed, the cell modes are generally more efficient than the closure mode, but it is less clear whether the optimized cell mode is better than the standard one: More local optimizations may be available with the optimized cell mode, but there is also more overhead associated with ensuring that the cell is always unevaluated.

In the next section we describe the choices we made in our compiler, and in Section 5.3 we describe other possibilities. Section 5.4 explains how conflicts are resolved when an expression that is assigned one interface is passed to a function that expects another interface.

5.2. *Our strategy*

Before generating target code, our compiler performs strictness analysis and path analysis, and then annotates each expression in the source program with a "safe" interface. The appropriate mode can often be determined using strictness analysis, which tells us whether it is safe to pass an argument to a function using value mode. All nonstrict parameters to functions are currently assigned cell mode, but we are evaluating heuristics for choosing between cell mode and optimized cell mode. In mutually recursive equation groups, we use strictness analysis to determine which functions or expressions will definitely be evaluated, and dependency analysis[9] to identify all circular dependencies. Then identifiers bound to functions or expressions that will definitely be evaluated and that have no circular dependencies are assigned value mode, while all others are assigned cell mode.

Determining the appropriate higher-order interface is a bit more complicated. Each primitive returns a value with a known interface, usually value mode and no higher-order behavior. Many primitives (such as "+") take only first-order arguments, which do not require interface information. An exception is the conditional, which is "polymorphic": for any interface i, the conditional has interface $V/(V \Rightarrow V/(i \Rightarrow V/(i \Rightarrow i)))$. That is, the conditional (written in curried form) is a value (in fact, a known value that will be compiled in line) that takes a predicate in value mode, a consequent with some appropriate interface i, an alternate with the same interface i, and returns a value with that same interface (recall that the arms of a conditional may return functions).

More complications arise with user-defined functions. When a "known" function is used, we use its assigned interface. When an "unknown" (e.g., lambda-bound) function is used, we compile it with a "standard" interface that is always safe, i.e., that does no more evaluation than we can prove necessary. There are two standard interfaces: i_{Cstd}, for use when the function value is assigned cell mode, and i_{Vstd}, for when the function value is assigned value mode. These are defined in a mutually recursive manner as follows:

$$i_{Cstd} = C/(i_{Cstd} \Rightarrow i_{Vstd})$$
$$i_{Vstd} = V/(i_{Cstd} \Rightarrow i_{Vstd})$$

Thus in a call to an unknown function, the argument is passed as a thunk (cell mode) and the result returned is evaluated. If the result of calling an unknown function is also a function, it too expects its arguments delayed using the cell mode.

5.3. *Alternative strategies*

Instead of using a "standard" interface for unknown functions, we could use a *collecting interpretation* [11] to determine the interfaces of all functions that could be

[9] This dependency analysis involves building a dependency graph and finding its strongly connected components; a description of the technique may be found in [16].

the unknown function (in other words, the unknown function becomes known). Thus, if an unknown function f is lambda-bound, we could find all applications of the function g to which f is a parameter and extract the interface information for the argument corresponding to f from each call. This information could be used in one of two ways:

1. We could take the "greatest lower bound" of the interfaces (assuming an ordering in which a better interface is stronger than a worse one), giving us the best interface that is safe for all calls to g. This works well when all interfaces are the same, or when only a small percentage of them are better than the others. However, if most of the possible interfaces are "good," e.g., $V/(V \Rightarrow V)$, but one application requires a "bad" interface, e.g., $V/(C \Rightarrow V)$, by this method all applications will be forced to use the bad interface.
2. We could compile n different versions for g, one for each of the possible interfaces. This works well in the situation described above with one bad interface because the majority of the applications are not penalized by the bad one. However, it has the disadvantage of potentially causing the target program size to grow exponentially in the depth of nesting of functions taking functions as arguments.

The main disadvantage of the collecting interpretation is that it is expensive to compute and the magnitude of the payoff is unclear.

5.4. Mode coercions

After each expression has been annotated with an appropriate interface, we generate code for the program. We may find that an expression to which we assigned interface i_1 is being passed to a function (or primitive) that expects arguments with interface i_2. In this case, we must generate code to coerce the expression from interface i_1 to interface i_2 at run time, which is performed as follows:

$$\textit{int-coerce}(m_1;m_2,exp) = \textit{mode-coerce}(m_1, m_2, exp)$$

$$\begin{aligned}
&\textit{int-coerce}\ ((m_1/(i_{11} \Rightarrow i_{12})), (m_2/(i_{21} \Rightarrow i_{22})), exp) = \\
&\qquad \text{let}\ \ \textit{old-call} = ((\mathbf{FORCE}_{m1}\ exp)\quad \textit{int-coerce}(i_{21}, i_{11}, \mathbf{x})) \\
&\qquad\qquad \textit{new-call} = \textit{int-coerce}(i_{12}, i_{22}, \textit{old-call}) \\
&\qquad \text{in}\ \ (\mathbf{TRIVIAL\text{-}DELAY}_{m_2}\ \ (\mathbf{lambda\ (x)}\ \ \textit{new-call}))
\end{aligned}$$

The first case occurs when exp is first-order; here i_1 and i_2 contain no higher-order information and may be represented by modes m_1 and m_2, and *int-coerce* reduces to *mode-coerce*, defined below.

In the second case, exp is a function-valued expression in mode m_1 that expects its argument using interface i_{11} and returns a value using interface i_{12}. However, the function to which *exp* is being passed expects its argument to be in mode m_2, to take an argument using interface i_{21} and return a value using interface i_{22}. Four coercions are required. From the inside out, we first force *exp*, and then apply it to a new identifier, **x**, which uses interface i_{21}, and must be converted to interface i_{11}. We then convert the result of this call from interface i_{12} to interface i_{22}. Finally, we wrap this call in a lambda, creating something with interface $V/(i_{21} \Rightarrow i_{22})$, which we (trivially) delay to mode m_2.

Coercion from mode m_1 to mode m_2 is achieved most generally as follows:

$$\textit{mode-coerce}(m_1, m_2, f) = (\textbf{DELAY}_{m2}\ (\textbf{FORCE}_{m1}\ f))$$

Note, however, that this may be optimized for certain mode pairs. For example,

$$\textit{mode-coerce}(V, m_2, f) = (\textbf{TRIVIAL-DELAY}_{m2}\ f)$$

Finally, when we generate code for $(\textbf{DELAY}_m\ \textbf{x})$ and $(\textbf{FORCE}_m\ \textbf{x})$, we can take advantage of the optimizations discussed in Section 4.

6. Conclusions

We have presented several representations for thunks as used in implementations of nonstrict functional languages, together with optimizations to reduce their space and time complexity. The analysis required for the optimizations occurs at the *source code* level; that is, we analyze the functional program itself (via path analysis) to determine how to generate code that handles thunks efficiently. An alternative would be to choose a model for thunks and then generate code using that model without optimization, relying on an analysis of the *target code* to improve performance of thunks. This has the possible advantage of being able to use more traditional compiler optimization techniques, but it is not clear how well these techniques would perform on any given representation for thunks. Furthermore, our method is more portable: Since the information is derived at the source level, it is available during code generation *regardless of the target language*. As discussed in the introduction, many virtual machines have been proposed for the implementation of functional languages, and optimizing the target code would require a new analysis for each virtual machine.

We depend on path analysis to gather the necessary order-of-evaluation information to guide code generation, and on strictness analysis to choose a thunk mode for each expression in a source program. While value mode is generally preferred when it is safe, it is not clear which of closure mode, cell mode, and optimized cell mode performs best, and to what extent this is program-dependent. Experience with a broad range of programs on a variety of machines will be necessary to make these decisions evident.

Acknowledgments

Thanks to Richard Kelsey for inspiring this work by pointing out that the closure representation of thunks could be improved, to Ken Traub for reminding us that we needed to do dependency analysis, and to John Hughes for his many helpful comments.

References

1. Abramsky, S., and Hankin, C. *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.
2. Bloss, A. *Path Analysis: Using Order-of-Evaluation Information to Optimize Lazy Functional Languages*, Ph.D. thesis, Department of Computer Science, Yale University, New Haven, CT, 1988.
3. Bloss, A., and Hudak, P. Path semantics. In *Proceedings, Third Workshop on the Mathematical Foundations of Programming Language Semantics*, Springer-Verlag LNCS, April 1987.
4. Burn, G. L., Hankin, C. L., and Abramsky, S. The theory of strictness analysis for higher order functions. In *LNCS 217: Programs as Data Objects*, Springer-Verlag, New York, 1985, pp.42–62.
5. Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Sym. on Prin. of Prog. Lang.*, ACM, 1977, pp. 238–252.
6. Fairbairn, J., and Wray, S. Tim: A simple, lazy abstract machine to execute super-combinators. In *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference*, Springer-Verlag LNCS 274, September 1987, pp. 34–45.
7. Hall, C. B., Wise, D. S. Compiling strictness into streams. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, ACM, January 1987, pp. 132–143.
8. Halstead, R. H., Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4) (1985) 501–538.
9. Henderson, P. *Functional Programming: Application and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1980.
10. Hudak, P. *ALFL Reference Manual and Programmer's Guide*. Research Report YALEU/DCS/RR-322, 2d ed., Yale University, New Haven, CT, October 1984.
11. Hudak, P., and Young, J. Higher-order strictness analysis for untyped lambda calculus. In *12th ACM Sym. on Prin. of Prog. Lang.*, January 1986, pp. 97–109.
12. Hudak, P., and Young, J. Collecting interpretations of expressions (without powerdomains). In *Proceedings of the 15th ACM Symposium on Prinicples of Programming Languages*, January 1988, pp. 107–118.
13. Hughes, J. Strictness detection in non-flat domains. In *LNCS 217: Programs as Data Objects*, Springer-Verlag, New York, 1986, pp. 42–62.
14. Johnsson, T. Efficient compilation of lazy evaluation. In *Proceedings of the SIG-PLAN 86 Symposium on Compiler Construction*, ACM, *SIGPLAN Notices* **19**(6) (June 1984), 58–69.
15. Jones, N. D., and Muchnick, S. S. Complexity of flow analysis, inductive assertion synthesis, and a language due to dijkstra. In *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1981, pp. 380–393.
16. Jones, S. P. *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, NJ, 1987.
17. Kieburtz, R. B. *The G-Machine: A Fast, Graph-Reduction Evaluator*. Technical Report CS/E-85-002, Dept. of Computer Science, Oregon Graduate Center, January 1985.
18. Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. Orbit: An optimizing compiler for Scheme. In *SIGPLAN '86 Symposium on Compiler Construction*, ACM, June 1986, pp. 219–233; published as *SIGPLAN Notices* **21**(7) (July 1986).
19. Liskov, B. Private communication, October 1987.

20. Mauny, M., and Suárez, A. Implementing functional languages in the categorical abstract machine. In *Proceedings 1986 ACM Conference on Lisp and Functional Programming*, Cambridge, MA, August 1986, pp. 266–278.
21. Mycroft, A. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings, of Int. Sym. on Programming*,Springer-Verlag LNCS Vol. 83, 1980, pp. 269–281.
22. The revised^3 report on the algorithmic language scheme. *SIGPLAN Notices* **21**(12) (December 1986), 37–79.
23. Turner, D. A. A new implementation technique for applicative languages. *Software—Practice and Experience* **9** (1979), 31–49.
24. Wadler, P., and Hughes, R. J. M. Projections for strictness analysis. In *Proceedings of 1987 Functional Programming Languages and Computer Architecture Conference*, Springer Verlag LNCS 274, September 1987, pp. 385–407.
25. Young, J. *Theory and Practice of Semantics-Directed Compiling for Functional Programming Languages*. Ph.D. thesis, Deparment of Computer Science, Yale University, New Haven, CT, 1988.
26. Young, J., and Hudak, P. *Finding Fixpoints on Function Spaces*. Research Report YALEU/DCS/RR-505, Department of Computer Science, New Haven, CT, November 1986.