Laziness Without All the Hard Work

Combining Lazy and Strict Languages for Teaching

Eli Barzilay
Northeastern University
eli@barzilay.org

John Clements
Northeastern University
clements@brinckerhoff.org

Abstract

Students have trouble understanding the difference between lazy and strict programming. It is difficult to compare the two directly, because popular strict languages and popular lazy languages differ in their syntax, in their type systems, and in other ways unrelated to the lazy/strict evaluation discipline.

While teaching programming languages courses, we have discovered that an extension to PLT Scheme allows the system to accommodate both lazy and strict evaluation in the same system. Moreover, the extension is simple and transparent. Finally, the simple nature of the extension means that the resulting system provides a rich environment for both lazy and strict programs without modification.

Categories and Subject Descriptors D.3.m [*Programming Languages*]: Miscellaneous

General Terms Languages

Keywords Lazy and Eager Evaluation, Teaching Programming Languages, PLT Scheme

1. Introduction

Computer science professors all over the world recognize the significance of the definitional interpreter as a central tool in the understanding of programming languages. In this approach, students understand the similarities and differences between programming languages by writing interpreters for these languages. These interpreters are structurally similar to formal specifications of the languages they define (the *defined* languages). As the course progresses, the students learn about new programming language constructs by adding corresponding rules to their interpreters. Since each interpreter is an extension of the prior one, they are typically all written in the same *defining* language.

This approach is natural and informative, and it is adopted in one form or another by many modern programming languages textbooks [1, 8, 11]. In fact, this approach follows directly from the maxim that "the best way to learn is to teach" and the observation that writing a program is exactly this: the programmer must teach the computer how to perform the given task, in the most detailed and pedantic fashion imaginable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FDPE'05 September 25, 2005, Tallinn, Estonia.
Copyright © 2005 ACM 1-59593-067-1/05/0009. . . $5.00.

The notion of a definitional interpreter is an old one. Reynolds [12] provides a synopsis of earlier work and is the starting point for much of the later work. In this paper, Reynolds classifies definitional interpreters based on two key features of the defining language: whether they permit higher-order functions, and whether they are call-by-value or call-by-name.

This classification adds a second axis to the space of definitional interpreters. Along with the features we are adding to the defined language, we must also consider the set of features in the defining language. Do we wish to change them, as well?

At first glance, the answer is "no". After all, we have observed already that extending an interpreter is possible only when the new and old interpreters are written in the same language. Changing the defining language could force students to re-implement their interpreters and needlessly disorient them.

Turning again to the question of how students learn, however, we see that while they gain experience in specifying the defined language, their experience in *developing* programs lies only in the defining language. Indeed, students may graduate from such a course without having written more than a few two-line programs in each of the languages defined. That is, the only programs they write are the test cases for their interpreters.

The clearest example of this problem is in the difference between *strict* and *lazy* languages. In a strict language, arguments to a function are reduced to values before calling the function. In a lazy language, however, arguments to a function are evaluated only when they are needed. So, for instance, a function which does not use its first argument will not cause that argument to be evaluated. This change is sufficiently fundamental that most students understand laziness only after writing many programs in a lazy language. Merely altering an existing interpreter to define a lazy language may not be enough to internalize the difference between strict and lazy languages.

Some programming texts address this through what they call "horizonal" integration, rather than the "vertical" integration of extending a single interpreter with different features. Specifically, they supplement their definitional interpreters with small programming assignments in a language that contains the desired features. So, for instance, the students might practice writing programs in a lazy language such as Haskell before modifying their interpreters to behave lazily. The problem with this approach is that the key difference—laziness—is buried in an avalanche of other differences. Changes in syntax and changes in type systems prove to be very large obstacles, particularly for beginning programmers. Instead, we would like a language that can behave either lazily or strictly without changes to any other part of the system. That is, laziness should be orthogonal to other features of the language.

We have discovered that this is possible, using the PLT Scheme framework. By changing the "language level" to one that we provide, students may evaluate the same expressions in a strict lan-

guage or in the corresponding lazy one. No changes whatsoever to the program text are required.

A combination of features makes this orthogonal switch possible. Scheme's syntax system [10] provides the tools needed to extend and alter the language, and PLT Scheme's module system provides the abstraction needed to make this change local, so that code written in the strict language is still evaluated eagerly.

A key advantage to this architecture is that PLT Scheme's existing facilities are available to both strict and lazy languages. This includes a rich set of libraries, and a variety of program tools, including a syntax checker, a coverage tester, and an error-tracing facility, among others [5, 3].

This paper has three more sections. In section 2, we show how the issue of laziness arises in a programming languages course, how our lazy language fits into the curriculum, and how the existing programming tools work without modification on the new language. In section 3, we show how the PLT scheme environment makes it possible to add laziness in a high-level way. Section 4 concludes.

2. Laziness in Action

To illustrate our extension, we consider a concrete example of its use. What follows is drawn from lectures given in Northeastern's Programming Languages course[1]. The course uses Krishnamurthi's textbook "Programming Languages: Application and Interpretation" [11]. In this approach, each new concept is

- introduced and analyzed in class,
- demonstrated in Scheme (the defining language),
- implemented as an extension of the defined language's interpreter,
- exercised at the defining level (usage) and the defined level (implementation).

Throughout the course, the students develop a series of interpreters whose complexity gradually increases.

Figure 1 shows the definition and the implementation of a simple language[2] that is demonstrated in the early stages of the course. Since the students have experience only with eager languages, they read this interpreter as the definition of an eager language, and they translate this belief to the formal definition as well.

This provides a natural entry for a discussion of lazy evaluation, and to explain that the evaluation rules for '`with`' and for '`call`' can be modified to operate in a lazy way,[3] which will change the defined language to a lazy one:

$$\text{eval}(\{\text{with } \{x\ E_1\}\ E_2\}) \ \rightarrow\ \text{eval}(E_2[E_1/x])$$
$$\text{eval}(\{\text{call}\}\ E_1\ E_1) \ \rightarrow\ \text{eval}(E_f[E_2/x]) \quad \text{if} \ldots$$

However, going back to the (apparently) eager version that was defined and implemented, we can see (as noted by Reynolds) that the defined language is eager only because our defining language is eager, and that in fact the formal definition is non-deterministic in this regard. Students have difficulty understanding this possibility, and assume that the definition given could only be that of a strict language.[4]

One possible approach is to make a quick detour and introduce Haskell [9] — a language that is considerably different from Scheme in both syntax and semantics. As mentioned above, we believe that this approach puts an additional burden on students, since Haskell differ from Scheme on many fronts on top of its choice of evaluation order. For a crowd of stronger students, this might work, but we believe that for the average student, the simultaneous changes may be distracting.

2.1 Our Solution: A Lazy Scheme

Before settling on a solution, we considered and discarded several alternative approaches, including the following:

- Implement an interpreter which students use as their defining language. This leads to a heavy performance hit, making it impossible for students to run anything more than toy programs in their interpreter.
- Have the students implement a lazy language, and then assign exercises to be implemented in their defined language. The main problem here is that students consider their defined language as a toy, so they will dismiss such exercises as no more than mere academic illustrations, and by association dismiss lazy evaluation as such.
- Avoid introducing a lazy language, and instead demonstrate some restricted laziness in the defining language. For example, use Scheme's '`delay`' and '`force`' to demonstrate some degree of laziness. While practical, the explicit nature of the abstraction prevents a deep understanding of the differences between lazy and eager evaluation.

We believe that actual programming experience is crucial for internalizing lazy programming. Switching languages makes it less accessible, and the above approaches avoid making students experience lazy programming first-hand.

In short, we need a practical implementation of a lazy variant of Scheme, which should be implemented as an extension of our existing language. As we shall see in the following section, there are several features that are unique to PLT Scheme which make it possible to define a "new language" with different semantics, yet have it be a well-behaved part of the same system. This means that we get the environment support of DrScheme, as well as functionality that exists in many libraries that are included in PLT Scheme. The lazy language is implemented as a module, so existing code that does not use this module is not affected. It is also possible to use standard code from a lazy program and vice versa, under certain conditions — procedures from normal Scheme modules are treated as strict primitives in lazy code, and values from lazy modules can contain delayed promises in strict code.

2.2 Examples

The Lazy Scheme language is bundled as a PLT package that is used in the course. (The interested reader can install it from `http://csu660.barzilay.org/csu660.plt`[5].) Once the package is installed, DrScheme's language selection dialog will have a new "CSU660 Lazy Scheme" entry which makes the definitions and interactions windows use the lazy language.

As a first example, we can enter some code and witness how only the parts that are required by interaction output is executed. By default, the Lazy Scheme language level uses DrScheme's syntactic coverage feature, which highlights code that is "touched" during evaluation. Figure 2 shows a DrScheme screenshot that demonstrates such an interaction[6].

As this demonstrates, constructors of lists ('`cons`', '`list`', '`list*`') and of other objects are properly lazy in the new language, and accessors are strict. This means that instead of dealing

[1] CSU660, `http://www.ccs.neu.edu/course/csu660/`

[2] This is Krishnamurthi's 'FWAE' language. Curly braces are used in defined languages to avoid confusing them with the defining language.

[3] A little later in the course we discuss name capturing.

[4] We imagine that students learning in Haskell would be similarly impaired, although in the other direction.

[5] Currently, this requires using version 209 of PLT Scheme.

[6] Coverage is indicated by colors, underlines added here for printout clarity.

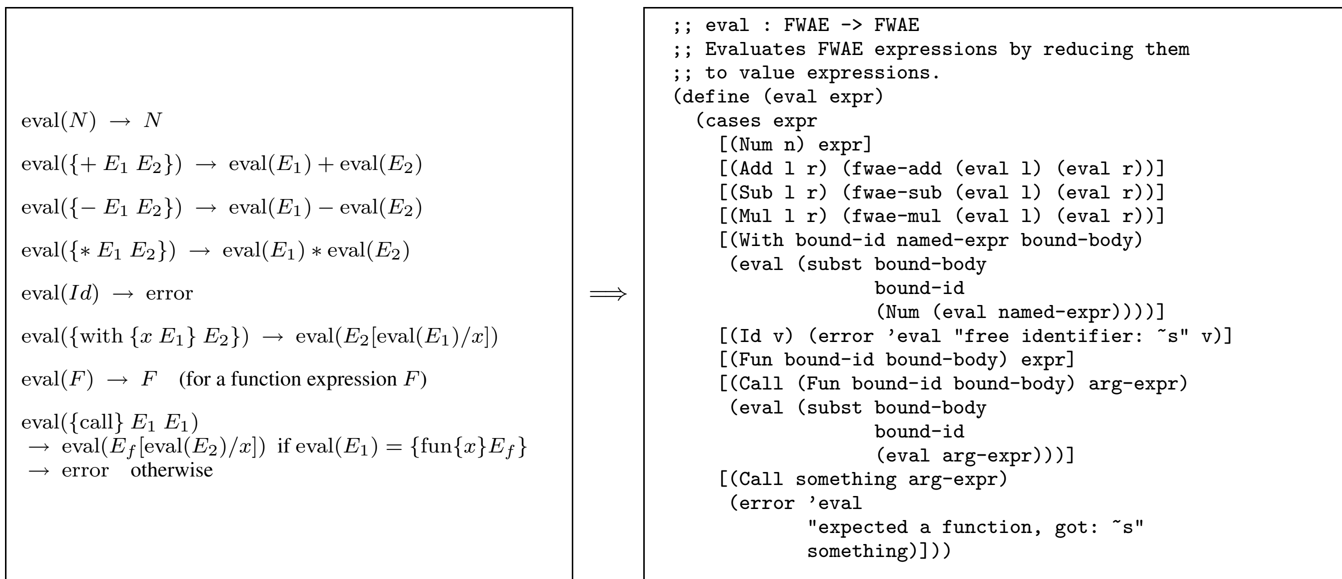

Figure 1. Definition and implementation of a simple language

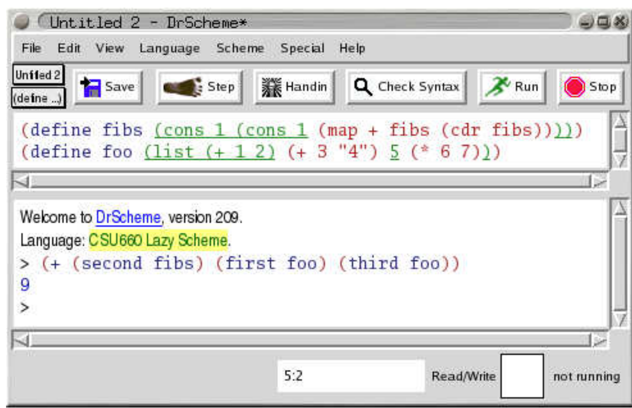

Figure 2. Demonstrating syntactic coverage in Lazy Scheme

```
(define nats (cons 1 (map add1 nats)))
(define (divides? n m)
  (zero? (modulo m n)))
(define (sift n l)
  (filter (lambda (x) (not (divides? n x))) l))
(define (sieve l)
  (cons (car l) (sieve (sift (car l) (cdr l)))))
(define (n-primes n) (take n (sieve (cdr nats))))
```

Figure 3. Using infinite lists in lazy Scheme

with special names for operations on streams [1, 2], we use known Scheme names: the language is the same, only the evaluation order changed. The code in Figure 3 demonstrates using infinite lists in plain Scheme syntax.

Finally, we can get back to Reynolds' observation, which is demonstrated effectively using our Lazy Scheme. Almost any of the interpreters that are implemented throughout the course, e.g., the code in Figure 1, can be used *as is* in the Lazy Scheme context to yield a lazy evaluator. Re-examining the code in Figure 1, reveals that there is a little more than plain Scheme to our interpreter. The '`cases`' expression is a syntactic extension that is used throughout the course, together with a new '`define-type`' declaration. '`define-type`' is used to define a type which is a disjoint union of a few record variants, and '`cases`' checks the type of its input and deconstruct it by pattern-matching. Together, they are roughly equivalent to using types in a statically typed (functional) language like ML. This functionality is implemented by some non-trivial syntactic code. It is essential to the coursework, so it must be present in the Lazy Scheme language as well; which is easily achieved by using the *same code* in the two contexts. This confirms the usability of the lazy language, since it is used with code that implement our teaching framework.

Figure 4 shows a more sophisticated evaluator. Once again, this code is valid in both languages, yielding an eager or a lazy evaluator.

3. Implementing a Lazy Scheme

Our lazy language implementation relies heavily on PLT Scheme's module system [7]. This system provides a robust way of defining modules that export both standard functionality and syntax transformations. The core of the lazy language delays all function applications, and forces arguments to strict functions — this is a known solution to the off-by-one problem that naive stream implementations suffer from (our solution is similar to even-style streams [13]). This is implemented by the following transformation:

```
(f x ...)
  -> (~ (let ([f (! f)])
          (if (lazy? f) (f x ...) (f (! x) ...))))
```

where '~' is '`delay`' and '`!`' is '`force`' iterated as many times as necessary to get a value. The rationale behind iterating '`force`' is that we avoid complex bookkeeping (e.g., SRFI-40 [2]) by treating all promises as delayed expressions. Actually, the Scheme Report mentions treating promises as the values they encapsulate as a viable implementation strategy [10, Section 6.4]: "It may be the case that there is no means by which a promise can be operationally distinguished from its forced value".

The principle is therefore simple; PLT Scheme has a combination of powerful features that makes it possible to implement this lazy language in a way that cooperates with the environment, so that strict and lazy code can be combined via the module system.

```
(define-type FLANG
  [Num  (n number?)]
  [Add  (lhs FLANG?) (rhs FLANG?)]
  [Sub  (lhs FLANG?) (rhs FLANG?)]
  [Mul  (lhs FLANG?) (rhs FLANG?)]
  [Div  (lhs FLANG?) (rhs FLANG?)]
  [Id   (name symbol?)]
  [With (name symbol?) (named FLANG?) (body FLANG?)]
  [Fun  (name symbol?) (body FLANG?)]
  [Call (fun-expr FLANG?) (arg-expr FLANG?)])

;; eval : FLANG env -> VAL
;; evaluates FLANG expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Add l r) (arith-op + (eval l env) (eval r env))]
    [(Sub l r) (arith-op - (eval l env) (eval r env))]
    [(Mul l r) (arith-op * (eval l env) (eval r env))]
    [(Div l r) (arith-op / (eval l env) (eval r env))]
    [(With bound-id named-expr bound-body)
     (eval bound-body
           (Extend bound-id (eval named-expr env) env))]
    [(Id v) (lookup v env)]
    [(Fun bound-id bound-body)
     (FunV (lambda (arg-val)
             (eval bound-body
                   (Extend bound-id arg-val env))))]
    [(Call fun-expr arg-expr)
     (let ([fval (eval fun-expr env)])
       (cases fval
         [(FunV proc) (proc (eval arg-expr env))]
         [else (error 'eval
                      "expected a function, got: ~s"
                      fval)]))]))
```

Figure 4. Parts of an evaluator code that can be used *as is* in both strict and lazy Scheme

The following is a list of these features and how they contribute to the implementation. The MzScheme language manual [6] describes these features in detail.

Primitive application syntax: The transformation that we use is needed for all function application forms. In most Scheme implementations, this requires implementing a code-walker that can identify and ignore special forms and macros and is able to deal with code that is generated by macros.

In PLT Scheme, however, all function applications are first expanded as uses of the special '`#%app`' syntax [6, Section 12.5]. Furthermore, it is possible to create a new 'language module' that can provide its own version of Scheme primitives, including the '`#%app`' syntax. Our lazy language module uses this to implement the transformation of application forms. Figure 5 shows the relevant part of the (simplified) code that implements the new '`#%app`' as well as a new '`apply`' function (the '`provide`' form is in charge of exporting a '`mzscheme`'-like language, except for new versions of '`#%app`' and '`apply`').

Note also that '`!`' is a function in the strict implementation, but it must be treated as a special form when it is used in the resulting lazy language or it will get delayed like other functions — strictness in a lazy language must be a special form [4].

Applicable records: For the implementation of our transformation we need to determine when a function is lazy. Obviously, known built-in constructors like '`cons`' and '`list`' are lazy, and non-constructor primitives are strict. But we cannot assume that all non-built-in functions are lazy or we would not be able to use Scheme functions from conventional modules imported as strict functionality.

The solution exercises PLT Scheme's ability to define new record types ('structs') that can be applied as functions. This

```
(module lazy mzscheme
  (define-syntax (~app stx)
    (syntax-case stx (!)
      ;; do not treat this as normal applications
      [(_ ! x) (syntax/loc stx (! x))]
      [(_ f x ...)
       (with-syntax
           ([(y ...) (generate-temporaries #'(x ...))])
         (~ (let ([p (! f)] [y x] ...)
              (if (lazy? p) (p y ...) (p (! y) ...)))))]))
  (define (~apply f . xs)
    (let ([f (! f)] [xs (!list (apply list* xs))])
      (apply f (if (lazy? f) xs (map ! xs)))))
  (provide (all-from-except mzscheme #%app apply)
           (rename ~app #%app)
           (rename ~apply apply)))
```

Figure 5. Implementing lazy function applications

can be used to annotate function values with source code, documentation, etc. We redefine '`lambda`' so it generates such tagged functions, making it possible to know when a function value was generated by lazy code. Checking for lazy functions is now simple: those that are tagged as lazy cover user-defined code and built-in constructors (which are re-provided as tagged values), record constructors are also considered lazy. All other functions are strict.

Technicalities: There are a few user-interaction technicalities that are specific for PLT Scheme. For example, setting a custom printer that forces (nested) evaluation results rather than have users force values they want to see.

Module system and syntax transformers: Finally, it is worth repeating that the resulting module cooperates with the rest of PLT Scheme — bindings from the two languages are not confused, and programs can be made of modules of both kinds without problems; many DrScheme tools "just work". Specifically, separate compilation works as expected even when modules are developed separately and later combined as parts of a single application. This is a good demonstration of the power of the PLT Scheme module system [7].

4. Conclusion

Our work demonstrates two things. First, that PLT Scheme's syntax and module systems make it possible to add such fundamental features as laziness to an existing language in a transparent and high-level way. Second, that such an extension has the crucial advantage that it inherits a wealth of libraries and environment tools.

As a result of these developments, it is now possible to show students in a programming languages course the difference between strict and lazy languages in isolation. That is, students can compare strict and lazy evaluations of the same program text. Furthermore, they can do so without giving up existing libraries, or their current set of tools.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] P. L. Bewig. SRFI 40: A library of streams. `http://srfi.schemers.org/srfi-40/`.

[3] J. Clements, M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. Fostering little languages. *Dr. Dobb's Journal*, March 2004. (Invited Paper).

[4] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[5] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A programming environment for Scheme. *Journal of Functional Programming*, 2001.

[6] M. Flatt. PLT MzScheme: Language manual. `http://www.plt-scheme.org/software/`, 1996–2005.

[7] M. Flatt. Composable and compilable macros: You want it *when?* In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[8] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, 2nd edition, 2001.

[9] P. Hudak and P. Wadler. Report on the programming language Haskell. Technical Report YALE/DCS/RR777, Yale University, Department of Computer Science, August 1991.

[10] R. Kelsey, W. Clinger, and J. Rees (Eds.). Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[11] S. Krishnamurthi. Programming languages: Application and interpretation. `www.cs.brown.edu/~sk/Publications/Books/ProgLangs/`, 2003–2005.

[12] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740. ACM Press, 1972.

[13] P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language, without even being odd. In *Workshop on Standard ML*, Baltimore, September 1998.