Dynamic Identifiers Can Be Neat

by

Bruce F. Duba, Matthias Felleisen
and Daniel P. Friedman

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT No. 220

Dynamic Identifiers can be Neat

by

B.F. Duba, M. Felleisen, & D.P. Friedman

April, 1987

This material is based on work supported in part by National Science Foundation grants number DCR 83-03325 and DCR 85-01277, and by an IBM Fellowship Award to Matthias Felleisen.

Dynamic Identifiers can be Neat

Bruce F. Duba, Matthias Felleisen, Daniel P. Friedman

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405, USA

Abstract

Linguistic facilities to express dynamic extent are common in modern, lexically-scoped Lisps. In such languages there is the traditional possibility of having identifiers with *dynamic scope*, or the alternative of having statically bound identifiers with *dynamic extent*. The latter possibility is superior since it inherits all the advantages of lexical scoping, however, the known implementation techniques interfere with the correct optimization of tail-recursion. We propose a new implementation strategy and show that it does not suffer from this problem.

1. Introduction

Static scoping is quickly becoming the standard for the Lisp family of programming languages, but many still include some abstraction for expressing dynamic extent. A prominent example is Common Lisp [9]. In Common Lisp it is possible to declare *special* variables. Special variables in Common Lisp have no textual restriction on *where* references may be made but are only accessible during the evaluation-time of the declaration body. This semantics for variables is the same as the one in traditional, dynamically-scoped Lisp [7].

Lexical scope gives the language designer a new alternative: he may now enforce dynamic *extent* on lexical variables. This has the advantage of only having one scoping mechanism in the language. All variables share the security and modularity of static scoping [3], while allowing the expression of dynamic behavior. An example of this approach is found in MIT Scheme [1, 4, 5]. MIT Scheme includes the special

form **fluid-let**. It is typically implemented as a syntactic extension:

$$\begin{aligned}(\textbf{fluid-let}\ (id\ val)\ exp) \equiv (\textbf{let}\ &(temp\ id)\\ &(\textbf{set!}\ id\ val)\\ &(\textbf{let}\ (res\ exp)\\ &\quad(\textbf{set!}\ id\ temp)\\ &\quad res)).\end{aligned}$$

Instead of creating a new binding for *id*, this form saves the current value of *id* in a temporary variable, then assigns *val* to *id* for the time of the evaluation of *exp*, reassigns *id* its original value, and finally returns the result value.

While the addition of lexical variables with dynamic extent to Lisp is quite common [1, 2, 4], it causes a serious problem. In such languages iterative algorithms are generally expressed as tail-recursive functions. The defining report on Scheme actually requires tail-recursion to be equivalent to iteration [8]. A proper implementation of tail-recursion implies that only the evaluation of subexpressions in applications may push information onto the control stack of the interpreter [10]. Thus, the invocation of the function

$$(\textbf{define}\ \text{loop}\ (\textbf{lambda}\ (x)\ (\textbf{let}\ (x\,x)\ (\text{loop}\ x))))$$

results in an infinite loop, that never causes a stack overflow. However, when we replace the **let** by a **fluid-let**

$$(\textbf{define}\ \text{loop}\ (\textbf{lambda}\ (x)\ (\textbf{fluid-let}\ (x\ x)(\text{loop}\ x))))$$

this is no longer true. The implementation of **fluid-let** above forces the interpreter to *remember* the reset-assignment of x after returning from the call to loop. We believe that this is unacceptable.

In this paper we present a tail-recursive realization of the **fluid-let** programming paradigm. It is based on the idea of a dynamic store, which in turn is derived from the concept of a dynamic environment. To this end, we formalize both dynamic scope and dynamic extent facilities in lexically scoped languages with a series of denotational-style [11] interpreters. We show that, contrary to expectations [4, 1, 5], *neither* of them causes any problems for the correct implementation of tail-recursion. In the next section the two basic object languages are defined. The third section contains the interpreters. In the fourth section we discuss the relationship

⟨expression⟩ ::=
  ⟨identifier⟩
| **(lambda (**⟨identifier⟩**)** ⟨expression⟩**)**
| **(**⟨expression⟩ ⟨expression⟩**)**

Figure 1a: Core of the Object Language

⟨expression⟩ ::=
  (dynamic-scope (⟨identifier⟩**)** ⟨expression⟩**)**
| **(dynamic-lookup** ⟨identifier⟩**)**

Figure 1b: The Scope-Language

⟨expression⟩ ::=
  (dynamic-extent (⟨identifier⟩**)** ⟨expression⟩**)**

Figure 1c: The Extent-Language

to other language constructs and show how they benefit from our implementation. The fifth section addresses specific implementation details.

2. The Object Language

For the purpose of this paper we restrict our attention to the purely functional subset of Lisp. The translation of our result into the framework of Common Lisp, Scheme, or other programming languages with similar facilities is straightforward.

The core of the object language consists of identifiers, **lambda**-abstractions, and applications. The concrete syntax is defined in Figure 1a. This core language has the usual semantics. An identifier is a placeholder for a value, a **lambda**-abstraction corresponds to a first-class, call-by-value procedure, and an application invokes the value of the function on the value of the argument.

We extend the core language in two different ways. Figure 1b defines the first extension. When a **dynamic-scope** expression is applied to a value, it binds its identifier to this value for the dynamic extent of the expression. An identifier that is bound with **dynamic-scope** must be accessed with **dynamic-lookup**. Thus, there is no need for shadowing rules as in Common Lisp [9, page 158]; a name may unambiguously be used as a lexical identifier and a dynamic identifier in the same expression.

The second extension is defined in Figure 1c. The identifier of a **dynamic-extent** expression must be bound by an enclosing **lambda**-abstraction. The application of a **dynamic-extent** expression temporarily replaces the value of the lexical identifier with the value of the argument. We refer to the two language extensions as the scope- and extent-language, respectively.

3. The Interpreters

We have formalized the semantics of the languages as denotational-style interpreters. First, we present an interpreter for the core language. The interpreter is a recursive function from expressions, environments, and continuations to answers. An environment is a map from identifiers to values and gives meaning to the free identifiers of an expression. A continuation is a function from values to answers, which represents the rest of the computation. It can be thought of as an abstraction of the control stack. By modeling control with continuations, it is easy to reason about such issues as the proper implementation of tail-recursion.

The interpretation of a program is syntax-oriented. For the identifier case the interpreter contains the following line:

$$\mathcal{E}[\![x]\!]\rho\kappa = \kappa(\rho[\![x]\!]).$$

Upon encountering an identifier the interpreter looks up its value in the current environment and passes this value to the rest of the computation.

A **lambda**-abstraction evaluates to a closure:

$$\mathcal{E}[\![(\textbf{lambda } (x)\ M)]\!]\rho\kappa = \kappa(\lambda v\kappa_v.\mathcal{E}[\![M]\!]\rho[x \leftarrow v]\kappa_v).$$

A closure is a functional object that encapsulates the **lambda**-body and the current, definition-time environment. When a closure is applied, it evaluates the function body in the closure-environment extended by a binding of the formal parameter to the value of the argument.

For an application (MN) the interpreter first evaluates M and then N, and finally invokes the resulting function closure on the argument value:

$$\mathcal{E}[\![(M\ N)]\!]\rho\kappa = \mathcal{E}[\![M]\!]\rho(\lambda m.\mathcal{E}[\![N]\!]\rho(\lambda n.mn\kappa)).$$

The appropriate sequencing is accomplished with the following continuations:

$$k_1 = \lambda m.\mathcal{E}[\![N]\!]\rho k_2 \quad \text{and} \quad k_2 = \lambda n.mn\kappa.$$

The first one, k_1, absorbs the value of M and goes on to evaluate the argument N; the second, k_2, performs the invocation after having received the value of N.

The defining equations of the interpreter have an interesting property: when a closure is applied, the continuation of the body of the closure is the same as the continuation of the application. In other words, when the evaluation of the function body is completed, the computation simply continues, no "cleanup" is necessary. We refer to interpreters with this property as *neat*. Neatness implies that tail-recursive function calls take place with no growth of the continuation. We would like the interpreters for the language extensions to share this property. Given this machinery, we can discuss the implementations of the language extensions.

3.1 An Interpreter for the Scope-Language

In the core interpreter lexical scoping is accomplished by having closures include the environment that was present at the time the **lambda**-expression was evaluated. For applications the current environment is used to evaluate the subexpressions of the application, but not the function invocation. In a dynamically scoped language the value of an abstraction need not include an environment, instead, it expects an environment when it is applied. Therefore, applications pass the application-environment to the invocation of the function. To mix the two scoping mechanisms, we use two environments.

The interpreter for the scope-language is a function from expressions, lexical environments, dynamic environments, and continuations to answers. The interpretation of a dynamically bound identifier is the same as the one for lexicals except that is uses the dynamic environment:

$$\mathcal{E}[\![x]\!]\rho\delta\kappa = \kappa(\rho[\![x]\!]),$$
$$\mathcal{E}[\![(\textbf{dynamic-lookup } x)]\!]\rho\delta\kappa = \kappa(\delta[\![x]\!]).$$

Both **lambda**- and **dynamic-scope**-expressions evaluate to closure objects:

$$\mathcal{E}[\![(\textbf{lambda } (x)\ M)]\!]\rho\delta\kappa = \kappa(\lambda v\delta_v\kappa_v.\mathcal{E}[\![M]\!]\rho[x \leftarrow v]\delta_v\kappa_v),$$
$$\mathcal{E}[\![(\textbf{dynamic-scope } (x)\ M)]\!]\rho\delta\kappa = \kappa(\lambda v\delta_v\kappa_v.\mathcal{E}[\![M]\!]\rho\delta_v[x \leftarrow v]\kappa_v).$$

These closures expect the application time dynamic environment as well as the value of the argument and the continuation of the application. When applied, a **lambda**-closure extends the definition-time lexical environment while a **dynamic-scope** closure extends the application-time dynamic environment.

In an application the dynamic environment is made available not only to the evaluation of the subexpressions but also to the function invocation:

$$\mathcal{E}[\![(M\ N)]\!]\rho\delta\kappa = \mathcal{E}[\![M]\!]\rho\delta(\lambda m.\mathcal{E}[\![N]\!]\rho\delta(\lambda n.mn\delta\kappa)).$$

It is obvious that this interpreter is still neat. The evaluation of the body of either a **dynamic-scope**- or a **lambda**-closure takes place in the continuation of the application, *e.g.*,

$$\mathcal{E}[\![((\textbf{dynamic-scope}\ (x)\ M)\ N)]\!]\rho\delta\kappa = \mathcal{E}[\![N]\!]\rho\delta(\lambda n.\mathcal{E}[\![M]\!]\rho\delta[x \leftarrow n]\kappa).$$

In particular, tail-recursive calls of **dynamic-scope** abstractions cause no growth of the interpreter's control structure.

3.2 The Extent-Language

In the extent-language **dynamic-extent** is a form of structured assignment. All variables are lexical, therefore there is no need for a dynamic environment. Instead, to formalize assignable variables, we must add a store component to the interpreter. An environment now maps identifiers to locations and a store maps locations to values. The new continuations represent the rest of the computation relative to some memory configuration. They take a value and a store and return an answer.

The definition of the store-based core interpreter is straightforward:

$$\begin{aligned}
\mathcal{E}[\![x]\!]\rho\sigma\kappa &= \kappa(\sigma(\rho[\![x]\!]))\sigma,\\
\mathcal{E}[\![(\textbf{lambda}\ (x)\ M)]\!]\rho\sigma\kappa &= \kappa(\lambda v\sigma_v\kappa_v.\mathcal{E}[\![M]\!]\rho[x \leftarrow n]\sigma_v[n \leftarrow v]\kappa_v)\sigma\\
&\qquad \text{where } n \notin \text{Domain}(\sigma_v),\\
\mathcal{E}[\![(MN)]\!]\rho\sigma\kappa &= \mathcal{E}[\![M]\!]\rho\sigma(\lambda m\sigma_m.\mathcal{E}[\![N]\!]\rho\sigma_m(\lambda n\sigma_n.mn\sigma_n\kappa)).
\end{aligned}$$

The value of an identifier is obtained by finding its location in the current environment and looking up the contents of that location in the store. When a closure is invoked, it allocates an unused location and binds the identifier to the location in the closure-environment, updating the store so that the location contains the value. In the application line, the two continuations for the evaluation of M and N expect a store. Thus, the interpreter ensures that the current store always reflects the true state of the memory.

The interpretation of **dynamic-extent** is derived from a transliteration of the

syntactic extension for **fluid-let**:

$$\begin{aligned}(\textbf{dynamic-extent}\ (id)\ exp) \equiv\ & (\textbf{lambda}(val)\\ & \quad (\textbf{let}\ (temp\ id)\\ & \quad\ \ (\textbf{set!}\ id\ val)\\ & \quad\ \ (\textbf{let}\ (res\ exp)\\ & \quad\quad (\textbf{set!}\ id\ temp)\\ & \quad\quad res))).\end{aligned}$$

The interpreter evaluates a **dynamic-extent** expression to a **dynamic-extent**-closure. The invocation of a **dynamic-extent**-closure updates the location associated with the identifier in the current store, runs the body, and finally resets the location in the resulting store to the value it had at application time:

$$\mathcal{E}[\![(\textbf{dynamic-extent}\ (x)\ M)]\!]\rho\sigma\kappa =$$
$$\kappa(\lambda v\sigma_v\kappa_v.\mathcal{E}[\![M]\!]\rho\sigma_v[\rho[\![x]\!] \leftarrow v](\lambda m\sigma_m.\kappa_v m\sigma_m[\rho[\![x]\!] \leftarrow \sigma_v(\rho[\![x]\!])]))\sigma.$$

This interpreter has the right semantics, but it is no longer neat! Instead of evaluating the body of a **dynamic-extent**-closure in the application-continuation, the interpreter constructs and uses a new continuation that accomplishes the necessary reassignment of the **dynamic-extent**-variable:

$$(\lambda m\sigma_m.\kappa_v m\sigma_m[\rho[\![x]\!] \leftarrow \sigma_v(\rho[\![x]\!])]).$$

The scope-language is able to express dynamic extent but does not have this problem. In the scope-interpreter the dynamic environment is not passed to continuations, because they already contain the correct dynamic environment for subexpression evaluation. If the store were handled in the same manner, there would be no need to undo assignments: the application-continuation would automatically access the correct store. This, however, would preclude the addition of traditional side-effects and would require a different implementation of **lambda**-closure application. The problem is that while two environments are used to model lexical and dynamic scope, the store must simultaneously implement both temporary and indefinite assignments. Therefore our solution is to split the store into a *main* store and a *dynamic* store.

The two stores have the same functionality: they take locations and return values. But, while the main store follows the flow of control in an application, the unchanged dynamic store is distributed to the evaluation of subexpressions:

$$\mathcal{E}[\![(MN)]\!]\rho\sigma\varphi\kappa = \mathcal{E}[\![M]\!]\rho\sigma\varphi(\lambda m\sigma_m.\mathcal{E}[\![N]\!]\rho\sigma_m\varphi(\lambda n\sigma_n.mn\sigma_n\varphi\kappa)).$$

Since the resetting of the **dynamic-extent**-variable is now unnecessary, the interpretation of **dynamic-extent** expressions is straightforward:

$$\mathcal{E}[\![(\textbf{dynamic-extent } (x)\ M)]\!]\rho\sigma\varphi\kappa = \kappa(\lambda v\sigma_v\varphi_v\kappa_v.\mathcal{E}[\![M]\!]\rho\sigma_v\varphi_v[\rho[\![x]\!] \leftarrow v]\kappa_v)\sigma,$$

however, identifier lookup is more complicated. First a check must be made to see if the location of the identifier has a value in the dynamic store. If so, that is the value; otherwise, the identifier is looked up in the main store:

$$\mathcal{E}[\![x]\!]\rho\sigma\varphi\kappa = \kappa((\rho[\![x]\!] \in \text{Domain}(\varphi) \rightarrow \varphi, \sigma)\rho[\![x]\!])\sigma.$$

The interpretation of **lambda**-expressions is basically unchanged:

$$\mathcal{E}[\![(\textbf{lambda } (x)\ M)]\!]\rho\sigma\varphi\kappa = \kappa(\lambda v\sigma_v\varphi_v\kappa_v.\mathcal{E}[\![M]\!]\rho[x \leftarrow n]\sigma_v[n \leftarrow v]\varphi_v\kappa_v)\sigma \quad \text{where } n \notin \text{Domain}(\sigma).$$

The new interpreter correctly realizes the semantics of the extent-language. The major difference is that *the new interpreter is neat*. It is also more sensitive to additional language features, that is, it simplifies the implementation of some, and also allows entirely new constructs that could distinguish it from the single store interpreter. We briefly investigate this phenomenon in the next section.

4. Additional features for the Extent-Language

There are two important features missing from the extent-language: assignments and labels. In a two-store world assignments could possibly change either store, but in the present framework a side-effect to the dynamic store could never be perceived: the dynamic store does not follow the flow of control. Thus the only feasible assignment operation in the current model modifies the main store:

$$\mathcal{E}[\![(\textbf{set-main-store!}\ x\ M)]\!]\rho\sigma\varphi\kappa = \mathcal{E}[\![M]\!]\rho\sigma\varphi(\lambda v\sigma_v.kv\sigma_v[\rho[\![x]\!] \leftarrow v]).$$

Traditionally, languages that have an operator like **dynamic-extent** allow side-effects to the dynamic store. To incorporate side-effects to the dynamic store, we would need to introduce another level of indirection. This is comparable to the division of environments into stores and environments for the introduction of traditional assignment.

Labels are another language facility that play an important role in conjunction with dynamic identifiers. In Common Lisp the respective operation is **catch**; in

Scheme, it is the primitive call-with-current-continuation. Both cause an enormous complication when added to the single-store language [5, 1, 4]. In particular, it must be guaranteed that any continuation that could possibly throw the computation outside of the **dynamic-extent** expression will do the reset-assignment to ensure the correct dynamic behavior. With two stores this becomes superfluous. Each continuation object already contains the correct dynamic store. Program continuations are practically equivalent to the interpreter continuations:

$$\mathcal{E}[\![(\textbf{catch } x\ M)]\!]\rho\sigma\varphi\kappa = \mathcal{E}[\![M]\!]\rho[x \leftarrow n]\sigma[n \leftarrow (\lambda v\sigma_v\varphi_v\kappa_v.\kappa v\sigma_v)]\varphi\kappa.$$

5. Implementation Notes

From the last interpreter for the extent-language it is clear that the only operation that may pay for the addition of dynamic-extent is identifier lookup. This may seem to be an unreasonable expense. However, if by static analysis or compiler directives, it can be proved that an identifier never has dynamic-extent, its lookup can be as cheap as those of traditional lexical identifiers. This observation is important because it means that we are not imposing a cost on those who do not use **dynamic-extent**.

An easy way to implement the dynamic store is as an association list contained in the control stack. When a location is to be given a new value in the dynamic store a new location value pair is pushed onto the stack. If the location already has a value it may be overwritten as long as it is not visible from any continuation. This gives automatic reclamation of the dynamic store and allows for a simple implementation of user continuations [6]. The association list method is usually abandoned in dynamically scoped languages because of the cost of lookups. However, the languages that we address are lexical by default, identifiers with dynamic extent are only used occasionally. The dynamic chain will tend to be short and rarely traversed. Paying a little more for dynamic lookup will not significantly degrade the performance of the whole system. It may seem as if the same kind of argument could be used by those promoting the re-assignment implementation: *since it is not often used, who cares if it is not properly tail-recursive.* There is an important difference between the costs of the two implementations. For programs that use dynamic extent heavily the worst that can happen using our method is that the program may run slowly. While the re-assignment implementation may fail for lack of control-stack space. In other words, correct programs may not run at all.

6. Conclusion

We have shown that the **dynamic-extent** facility or equivalently **fluid-let** need not interfere with the correct implementation of tail-recursion. We have also argued that the cost of providing **dynamic-extent** is not great and that it is well justified.

Also of interest is the use of denotational interpreters. Denotational semantics are typically used to address mathematical properties of programing languages such as: does the compiler preserve the functionality of a procedure definition. We believe that it is also appropriate to use denotational descriptions of languages to study the details of different implementation strategies.

References

1. DYBVIG, R. K. *The Scheme Programming Language*, Prentice Hall, Engelwood Cliffs, New Jersey, 1987.
2. FRIEDMAN, D.P., C.T. HAYNES, E. KOHLBECKER, M. WAND. Scheme 84 Interim Reference Manual, Technical Report No. 153, Indiana Univeristy Computer Science Department, 1985.
3. GORDON, M.J. *The Denotational Description of Programming Languages*, Springer-Verlag, New York, 1979.
4. HANSON, C., J.LAMPING. Dynamic binding in Scheme, unpublished manuscript, 1984, MIT.
5. HAYNES, C., D.P. FRIEDMAN. Embedding continuations in procedural objects, *TOPLAS*, 1987, to appear.
6. KRANZ, D., ET AL. ORBIT: An optimizing compiler for Scheme, *Proc. SIGPLAN '86 Symp. Compiler Construction, SIGPLAN Notices* **21**(7), 1986, 219–233.
7. MCCARTHY, J. *et al. Lisp 1.5 Programmer's Manual, Second Edition*, MIT Press, Cambridge, 1965.
8. REES J., W. CLINGER. (Eds.) The revised^3 report on the algorithmic language Scheme, *SIGPLAN Notices* **21**(12), 1986, 37–79.
9. STEELE, G. *COMMON LISP—The Language*, Digital Press, 1984.
10. STEELE, G.L. Lambda: The ultimate declarative, Memo 379, MIT AI Lab, 1976.

11. STOY, J.E. *Denotational Semantics: The Scott-Stratchey Approach to Programming Languages*, The MIT Press, Cambridge, Massachusetts, 1981.