Dynamic Variables

David R. Hanson and Todd A. Proebsting
Microsoft Research
1 Microsoft Way
Redmond, WA 98052
{drh,toddpro}@microsoft.com

ABSTRACT

Most programming languages use static scope rules for associating uses of identifiers with their declarations. Static scope helps catch errors at compile time, and it can be implemented efficiently. Some popular languages—Perl, Tcl, TeX, and Postscript—offer dynamic scope, because dynamic scope works well for variables that "customize" the execution environment, for example. Programmers must simulate dynamic scope to implement this kind of usage in statically scoped languages. This paper describes the design and implementation of imperative language constructs for introducing and referencing dynamically scoped variables—dynamic variables for short. The design is a minimalist one, because dynamic variables are best used sparingly, much like exceptions. The facility does, however, cater to the typical uses for dynamic scope, and it provides a cleaner mechanism for so-called thread-local variables. A particularly simple implementation suffices for languages without exception handling. For languages with exception handling, a more efficient implementation builds on existing compiler infrastructure. Exception handling can be viewed as a control construct with dynamic scope. Likewise, dynamic variables are a data construct with dynamic scope.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic scope, control structures.*

General Terms

Languages.

1. INTRODUCTION

Nearly all modern programming languages use static (or lexical) scope rules for determining variable bindings. Static scope can be implemented very efficiently and makes programs easier to understand. Dynamic scope is usually associated with "older" languages; notable examples include Lisp, SNOBOL4, and APL. Despite the prevalence of static scope, several widely used "newer" languages—PostScript, Tcl, TeX, and Perl—either use dynamic scope or include features that amount to dynamic scope. For example, Tcl's **`upvar`** can be used to access local variables in other active procedures, and Perl supports both static and dynamic scope on a per-variable basis. Shell environment variables are another example. Admittedly, these languages are used mostly in specific application domains, but their wide use suggests that dynamic scope is a useful facility.

Dynamic scope is particularly useful in applications and libraries that operate in a customizable "environment." Full-featured GUI packages are good examples; they support a plethora of style and presentation features, which must be programmed in some way. Even many "ordinary" applications have some of these characteristics; for example, compilers often have options that permit clients to specify target machines, compiler passes, linker options, etc., and compilers in integrated development environments typically have more options, because their "clients" are other programs.

Without dynamic scope, programming these kinds of features must be done explicitly in terms of the implementation language. Usually, objects are loaded up with fields or methods that implement these features, or methods and functions have zillions of parameters (perhaps with defaults). Indeed, an argument for optional, keyword-style formal parameters with defaults is that they cater to this kind of programming problem. In extreme cases, programmers move this boatload of fields or parameters into an object, which is passed as a parameter nearly everywhere. Methods are called to read, write, save, and restore these data. This popular approach usually simplifies the code considerably. It is also tantamount to implementing a set of variables with dynamic scope.

These unstructured, ad hoc approaches, which are often not type-safe and perhaps inefficient, can be replaced with equivalent language constructs that are structured, type-safe, and efficient. For example, Lewis et al. [6] extended a statically scoped functional language with "implicit parameters," which are dynamically scoped variables, and syntax for binding them and referencing them.

The remainder of this paper describes new imperative language constructs for supporting variables with dynamic scope. Like exception handling, such "dynamic variables" are best used sparingly, and the particularly simple design caters to this anticipated usage pattern. The simple implementation described below is probably sufficient for most languages, but a more sophisticated—and hence more efficient—implementation builds on the existing implementations of exception handling. Consequently, compilers for modern languages already have most of the infrastructure necessary for implementing dynamic variables.

2. DESIGN

Only two statements are necessary to add dynamic variables to most languages—one that creates and initializes a dynamic variable and one that binds a local variable to an instance of a dynamic variable.

The **`set`** statement instantiates and initializes a dynamic variable:

`set` *id* **`:`** *T* **`=`** *e* **`in`** *S*

The dynamic variable *id* is instantiated and initialized to the value of the expression, *e*, and the statement *S* is executed. The *id* is visible within *S* and may be read and written within *S*. The type of *e* must be a subtype of *T* or be assignable to identifiers of type *T*. The lifetime and scope of *id* is restricted to *S*; when *S* terminates, the dynamic variable is destroyed.

The **`use`** statement accesses a dynamic variable introduced by a **`set`** statement:

`use` *id* **`:`** *T* **`in`** *S*

The local variable *id* is bound by reference to a dynamic variable and the statement *S* is executed. The scope of the local introduced by the **`use`** statement is restricted to *S*, and *id* may be read and written within *S*. The local *id* is bound to the most recently created dynamic variable id_k with the lexographically identical name and for which T_k is a subtype of *T*. In Java terms, *id* is bound to the first id_k such that "*id*" = "id_k" and id_k **`instanceof`** *T* is true. For example:

```
set x:T3 = … in
  set x:T2 = … in {
    …
    use x:T1 in { … }
    …
  }
```

If **`T2`** and **`T3`** are both subtypes of **`T1`** in these fragments, the **`x`** in the **`use`** statement is bound to the **`x:T2`** in the nested **`set`** statement. If an appropriate dynamic variable is not found, the exception **`VariableNotFound`** is raised. In languages without exceptions, a runtime error occurs.

Types are considered in binding the identifier in a **`use`** statement because doing so is perhaps the only way to insure that existing code continues to work in a component-based system. If the lookup was based only on lexographical names, programmers would have to know the names of *all* of the other dynamic variables in a program, which is impossible when software is constructed from components. Likewise, if the lookup was based on exact types, instead of subtypes, existing **`use`** statements could not cope with new code in which the identifiers in **`set`** statements have been extended by declaring them subtypes of the original types.

A single **`set`** statement can contain multiple declarations, which are evaluated sequentially. That is,

`set` id_1 **`:`** T_1 **`=`** e_1**`,`** id_2 **`:`** T_2 **`=`** e_2**`,`** …**`,`** id_n **`:`** T_n **`=`** e_n **`in`** *S*

is equivalent to

```
set $id_1$ : $T_1$ = $e_1$ in
  set $id_2$ : $T_2$ = $e_2$ in
    set … in
      set $id_n$ : $T_n$ = $e_n$ in $S$
```

Likewise,

`use` id_1 **`:`** T_1**`,`** id_2 **`:`** T_2**`,`** …**`,`** id_m **`:`** T_m **`in`** *S*

is equivalent to

```
use $id_1$ : $T_1$ in
  use $id_2$ : $T_2$ in
    use … in
      use $id_m$ : $T_m$ in $S$
```

3. APPLICATIONS

Dynamic variables can replace the plethora of extra parameters and globals that often appear in, for example, GUI applications. Dynamic variables cut considerable verbiage in functions that use a few or none of them, because only those that are actually used appear in **`use`** statements.

Compiling loops and switch statements in recursive-descent compilers or in abstract-syntax tree traversals provide another simple example. Typical implementations associate a label or other handle with each loop and switch statement, and these data are used in compiling loop and switch exits, such as break and continue statements. Parameters for these data are passed to every recursive function in which the exit constructs can occur. For example, in lcc [2], every statement parsing function has three extra parameters, and most of those functions simply pass these parameters along to the functions they call. Using dynamic variables for these data eliminated these parameters and confined the use of these data to the functions for break and continue statements, and for case labels.

Dynamic variables were conceived during the design and implementation of Minicon, a new object-oriented, C++ implementation of the Icon programming language [4].

Minicon is based on Jcon, an interpretative implementation of Icon written in Java [9]. Icon includes numerous built-in keywords and functions that access implementation information. For example, Icon's string-scanning facilities uses the keywords **`&subject`** and **`&pos`** hold the current string being examined and the position in that string; built-in functions read and write these values. Likewise, **`&input`**, **`&output`**, and **`&errout`** give the default files for the built-in I/O functions.

The initial implementation of Minicon used ad hoc techniques for accessing keywords and other internal values. The most prevalent technique passed an "environment" pointer to those built-in functions that accessed keywords. Environments held a table of keywords and a cache of the values of a few frequently accessed keywords. This approach is essentially an implementation of dynamic scope. The result wasn't pretty: There were two ways to call built-in functions—with and without the environment pointer—so two data types represented built-ins. C++ derived classes and virtual functions hid some of this verbiage, but the clutter made the implementation harder to understand and to modify and complicated the interpreter.

These techniques were replaced with dynamic variables, using macros based on those shown in Appendix B. This change eliminated excess parameters to many functions and many lines of code, and thus cleaned up much of the original mess. Dynamic variables simplified the implementation and representation of built-in functions by reducing the number of ways to call them and the number of data types. The result is, unsurprisingly, easier to understand and to modify, and the performance cost is miniscule.

Perhaps a more important use of dynamic variables is in multithreaded applications. Languages and systems that support threads often provide mechanisms for specifying per-thread global variables; that is, variables with global scope and with lifetimes associated with the lifetimes of individual threads. These facilities usually require operating-system support. For example, the Microsoft C/C++ compiler supports "thread-local" global variables by making the appropriate Windows system calls. Unfortunately, this facility doesn't work properly in libraries that are loaded at runtime—thread-local variables are not initialized properly [10]. For those libraries, programmers must call the Windows system calls explicitly.

Dynamic variables are allocated on the stack, so they are automatically "thread-local," assuming a thread-safe implementation. Specifying them in a **`set`** statement in the thread's initial function makes them available (via **`use`** statements) to all functions called in the same thread, as if they were global variables.

Lewis et al. [6] provide a compelling survey on how dynamic scope helps make functional programs more clear and concise.

4. IMPLEMENTATION TECHNIQUES

For languages without exception handling, a simple, reasonably efficient, implementation of dynamic variables can be used. This implementation is also an operational semantics for the **`set`** and **`use`** statements. For languages that must support exception handling, the implementation of dynamic variables can use much of the existing exception handling implementation infrastructure.

4.1 Simple Implementation

Conceptually, the **`set`** statement creates a set of dynamic variables and pushes it onto a global stack of such sets. Reaching the end of the **`set`** statement pops the stack. The **`use`** statement searches the sets from the top of the stack down for each identifier listed in **`use`** statement and, if the identifier is found, stores the address of the dynamic variable in a local variable of the same name. Within the **`use`** statement, references to dynamic variables are compiled into the appropriate indirections.

The **`set`** statement can be implemented with *no* heap allocation overhead, because all of the allocations can be done at compile time as local variables. The stack of sets is simply a list of structures defined by the following pseudo-C code, one for each dynamic variable.

```
struct dVariable {
  struct dVariable *link;
  const char *name;
  Type *type;
  void *address;
} *current = 0;
```

`dVariable` instances are linked via the **`link`** field, the **`name`** field points to the name of the variable, the **`type`** field points to a type descriptor sufficient for testing the subtype relation, and the **`address`** field holds the address of the variable. A per-thread global variable, **`current`**, points to the head of the list of **`dVariable`**s.

The **`set`** statement

`set` id_1 **:** T_1 **=** e_1**,** id_2 **:** T_2 **=** e_2**,** …**,** id_n **:** T_n **=** e_n **`in`** S

is translated into the following compiler-generated code. The **`set`** statement declares a local variable for each listed *id*, and "pushes" a **`dVariable`** onto the **`current`** list for each *id*. After executing *S*, the previous value of **`current`** is restored from the **`link`** field of the first listed *id*. In the code below, **`typeof (`***T***`)`** returns a pointer to the type descriptor for *T*, and compiler-generated names are used where necessary.

```
{
  T_1 id_1 = e_1;
  struct dVariable dVar_id_1;
  dVar_id_1.name = "id_1";
  dVar_id_1.type = typeof (id_1);
  dVar_id_1.address = &id_1;
  dVar_id_1.link = current;
  current = &dVar_id_1;
  T_2 id_2 = e_2;
  struct dVariable dVar_id_2;
  dVar_id_2.name = "id_2";
  dVar_id_2.type = typeof (id_2);
  dVar_id_2.address = &id_2;
  dVar_id_2.link = &dVar_id_1;
  current = &dVar_id_2;
  …
  T_n id_n = e_n;
  struct dVariable dVar_id_n;
  dVar_id_n.name = "id_n";
  dVar_id_n.type = typeof (id_n);
  dVar_id_n.address = &id_n;
  dVar_id_n.link = &dVar_id_{n-1};
  current = &dVar_id_n;
  S
  current = dVar_id_1.link;
}
```

All the variables declared in the generated code fragment above are locals, so their storage is allocated as part of the stack frame. On 32-bit machines, the stack space overhead is 16*n* bytes for a `set` statement with *n* identifiers, not counting the string space for the identifier names. The only runtime overhead is the code for fetching `current` and its fields and the assignments to the fields and to `current`. A similar technique is used to build the “shadow stack” in the debugger cdb [5].

The `use` statement generates a linear search starting at `current`. The `use` statement declares a local variable for each listed *id* and searches for the appropriate dynamic variable instantiated by a `set` statement. The generated code below assumes that strings for the identifier names have been “internalized” so that there is only one copy of each distinct string. Consequently, two strings are identical if their addresses are equal. Internalization can be done at program startup or even at link time. The statement

`use` id_1 `:` T_1`,` id_2 `:` T_2`,` …`,` id_m `:` T_m `in` *S*

is translated into the following compiler-generated code:

```
{
  T_1 *id_1 = dSearch("id_1", typeof (T_1));
  T_2 *id_2 = dSearch("id_2", typeof (T_2));
  …
  T_m *id_m = dSearch("id_m", typeof (T_m));
  S
}
```

where `dSearch` is:

```
void *dSearch(const char *name,
    Type *type) {
  struct dVariable *p = current;
  for ( ; p != 0; p = p->link)
    if (p->name == name
    && type is a subtype of p->type)
      return p->address;
  raise VariableNotFound;
}
```

By design, the search occurs only once for each identifier at the entry to the `use` statement; multiple references to the id_k in *S* are compiled into simple indirections through the pointers shown in the generated code above.

This simple implementation is easily encoded into macros or function calls that provide most of the benefits of dynamic variables in existing languages, such as C and C++. As an example, Appendix B gives macro definitions for use in C++; these are used in the Minicon implementation, as detailed above.

More sophisticated search techniques could be used, but they would undoubtedly require more sophisticated data structures and thus complicate code generation for both `set` and `use` statements. For example, hashing would make `use` statements more efficient at the expense of `set` statements. Hashing identifiers can be done at compile time, so the `dVariable`s could be linked onto the heads of the appropriate chains with code as simple as shown above, assuming the size of the hash table is also known at compile time. At the end of the `set` statement, however, the `dVariable`s must be unlinked from the chains in the reverse order in which they were linked. The compiler could determine which `dVariable`s were the first ones linked onto each chain and unlink just those, but the savings achieved by this extra effort is likely to be small. Appendix C gives details. Another possible improvement that maintains the simplicity of the implementation above is to replace the calls to `dSearch` with code that searches for all *m* identifiers at once. Of course, the asymptotic complexity of these improvements is no better than the simpler implementation, but the average case might be improved, because failing searches are unexpected.

4.2 Novel Implementation

Most modern programming languages include exception-handling facilities, and typical implementations of these facilities are quite efficient. The runtime overhead when exceptions are not raised is small or zero; it’s all in the generated code and runtime library code that is executed when an exception occurs, under the assumption that exceptions are rare [1].

Dynamic variables can enjoy similar efficiencies by extending the mechanisms that a compiler for a modem language already implements. Java uses a typical implementation of exception handling [7]. The Java compiler emits exception tables, which allow the runtime system to identify exception-handling code when an exception occurs. These tables take only space. For example, given the following Java code, taken from Reference [7],

```
void CatchOne() {
  try {
    tryItOut();
  } catch (TestExc e) {
    handleExc(e);
  }
}
```

the compiler emits four items into an exception table entry: the starting and ending locations in the generated code for the **try** statement, the location of the exception handler, which is the call to **handleExc** in this example, and the type of the exception, i.e., **TestExc** above. Armed with the current location counter and the stack, the runtime system interprets these tables to find the appropriate exception handler, unwinding the stack as it goes, and transfers control to that handler.

Dynamic variables need similar data: the boundaries of a **set** statement, and the types, names, and addresses of the local identifiers. The table entries could thus be defined by:

```
struct dEntry {
  void *from;      // start of PC range
  void *to;        // end of PC range
  Type *type;      // declared type
  const char *name;
  int offset;      // local's frame pointer offset
}
```

The **from** and **to** fields give the boundaries of the relevant part of the **set** statement. The **type** and **name** fields are the same as the **dVariable** fields by the same names. The **offset** field is the offset from the frame pointer or other known location in a stack frame to the local variable id_k and is used to compute the runtime address of id_k. For the sake of explanation, the following structure models stack frames.

```
struct frame {
  struct frame *caller;
  void *retaddress;
  struct dEntry *dTable;
  …
}
```

Given a frame for a function, the **caller** field points to the caller's frame, **retaddress** holds the return address in the function's *caller*, and **dTable** points to an array of **dEntry** structures that identify the **set** statements in that function; this array is terminated with a **from** value equal to zero. Addresses of local variables are computed by adding their offsets to the address of the frame.

In actual implementations, the **dTable** field would more likely be computed directly from the location counter instead of being stored in the frame in order to avoid the initialization costs. The salient detail is that the appropriate table can be found given frame pointer and an address in the corresponding function. Similar comments apply to finding exception tables.

The **set** statement is compiled into a sequence of *n* assignments and *n* table entries as shown in Figure 1. As for exception-handling tables, the entries in the dynamic variable tables must be ordered so that the location counter ranges for nested **set** statements appear first.

dLookup searches the dynamic variable tables in the call stack for the first variable of a given name and type. The function **_self** returns a pointer to the caller's stack frame.

```
{
              T_1  id_1 = e_1;
start_1:      T_2  id_2 = e_2;
              …
start_{n-1}:  T_n  id_n = e_n;
start_n:      S
end:
}
```

from	to	type	name	offset
start$_1$	**end**	T_1	"id_1"	$offset_1$
start$_2$	**end**	T_2	"id_2"	$offset_2$
…				
start$_n$	**end**	T_n	"id_n"	$offset_n$
0				

Figure 1. Generated code and table for the **set** statement.

```
void *dLookup(const char *name,
    Type *type) {
  struct frame *fp = _self()->caller;
  void *pc = _self()->retaddress;
  for ( ; fp != 0; pc = fp->retaddress,
                   fp = fp->caller)
    if (fp->dTable != 0) {
      struct dEntry *p = fp->dTable;
      for ( ; p->from != 0; p++)
        if ( pc >= p->from && pc < p->to
        && p->name == name
        && type is a subtype of p->type)
          return (char *)fp + p->offset;
    }
  raise VariableNotFound;
}
```

The `use` statement is compiled into code that is nearly identical to the code shown in previous subsection, except that `dSearch` is replaced by `dLookup`:

```
{
  T_1 *id_1 = dLookup("id_1", typeof (T_1));
  T_2 *id_2 = dLookup("id_2", typeof (T_2));
  …
  T_m *id_m = dLookup("id_m", typeof (T_m));
  S
}
```

Implementing dynamic variables is actually simpler than implementing exception handling, because there are no control-flow or stack unwinding issues, which simplifies optimizations and debugging.

5. DISCUSSION

Much prejudice against dynamic scoping can be traced back to early LISP systems, which suffered from slow implementations and the "downward funarg problem" [11]. As we've shown, dynamic variables need not be slow, and the funarg problem is not at issue, because it affects only languages with closures.

Lewis et al. [6] give a compelling description of the benefits of dynamic scoping in a functional language. They do so by implementing dynamic variables as *implicit* parameters, which can be inferred from the underlying code. To infer implicit parameters, they rely on lexically distinguishing implicit parameters from ordinary identifiers. Our work differs from theirs in several ways. We propose alternative implementation strategies to provide implicit parameters. Our scheme does not rely on any inference mechanism to determine where dynamic variables might be needed. While their scheme nicely reflects their functional-language infrastructure, our scheme's dynamic variable binding mechanism reflects the common exception-handling facilities found in object-oriented languages. After the use of implicit parameters improved samples of code, they assert, "The resulting code is of a conciseness that is difficult to achieve when working in C or C++" (page 115). We agree completely, which is why we propose our dynamic scoping extensions for exactly that class of languages.

Lewis et al. [6] avoid the thorny issue of failing searches by doing whole-program analysis to detect missing dynamic variables at compile time. While this approach is strictly more robust than ours, and it is certainly possible to use in an imperative language, it is impractical for component-based software where the entire program is not available—and cannot ever be known.

The `set` and `use` statements are—intentionally—a minimal facility. They are ideal for the relatively infrequent use of dynamic variables, which, like exceptions, are best used in small doses. Reasonable improvements and enhancements are easy to imagine, but it is difficult to test their ultimate value. For example, separate variables seem superfluous when there's already a suitable local. Thus

set *id* **in** *S*

could abbreviate the idiom

id: *T*; …; **set** *id*: *T* = *id* **in** *S*

Likewise, name conflicts might plague `use` statements, particularly in component-based software. Specifying an alias could solve this problem:

use *id* : *T* **as** *id′* **in** *S*

Within S, the dynamic variable *id* would be accessed as *id′* in order to avoid conflicts with other uses of *id* in the context in which the `use` statement appears. This extension is similar to the way Modula-3 provides aliases for imported interfaces [8].

Finally, object-oriented languages usually include predicates for testing types, e.g., Java's `instanceof`, which suggests that a similar predicate for testing the existence of dynamic variables might be useful, e.g., **isdynamic(***id*, *T***)** would return true if *id* is a dynamic variable of type *T* or a subtype of *T* and false otherwise.

Dealing with failing searches in `use` statements is perhaps the weak point of our design. Such failures are bound to occur, and robust software components must anticipate and cope with them.

Catching the `VariableNotFound` exception and `isdynamic` are the only mechanisms in the present design for dealing with missing dynamic variables. For example, suppose a code generator is to emit its output to the `Stream` given by the value of the dynamic variable `output`. If `output` isn't defined, the output should go to the default stream, `stdout`. Writing this code with either `isdynamic` or a `try`-`catch` statement is awkward:

```
if (isdynamic(output, Stream))
  use output: Stream in S
else {
  output: Stream = stdout;
  S
}
```

or

```
try {
  use output: Stream in S
} catch (VariableNotFound) {
  output: Stream = stdout;
  S
}
```

It is, of course, possible to avoid the duplication of *S* by using additional variables, but the result remains awkward at best. The more important issue is whether or not to instantiate **`output`** as a dynamic variable when the search fails. The code fragments above do not; the following fragment does:

```
try {
  use output: Stream in S
} catch (VariableNotFound) {
  set output: Stream = stdout in S
}
```

Missing dynamic variables may occur frequently enough to warrant syntactic support. One possibility is to extend the **`use`** statement with a default expression:

`use` *id* **:** *T* **=** *default* **`in`** *S*

If *id* is found, *default* is ignored and *S* is executed; otherwise, *id* is initialized to *default* and *S* is executed. There two possible implementations of this form of **`use`** corresponding to the alternatives described above:

```
try {
  use id : T in S
} catch (VariableNotFound) {
  id : T = default;
  S
}
```

or

```
try {
  use id : T in S
} catch (VariableNotFound) {
  set id : T = default in S
}
```

The second alternative has the surprising effect of making a **`use`** statement behave as a **`set`** statement when the search fails. And both alternatives complicate considerably the implementations described in Section 4. Much more experience with dynamic variables would help solidify a final design.

Judging by the significant new languages introduced in the past decade, the programming language community has embraced exception handling as a mandatory language feature, because it helps build reliable and adaptable software. It is curious that only a *control* construct based on dynamic "scope" has found wide acceptance. Dynamic variables are a *data* construct with similar semantics, and, if incorporated into modern languages, the might prove to be a similarly important language facility.

6. REFERENCES

[1] Chase, D. R. Implementation of exception handling, Part I. *The Journal of C Language Translation* **5** (4), 229–40. June 1994.

[2] Fraser, C. W. and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Menlo Park, Calif.: Addison-Wesley. 1995.

[3] Gosling, J., B. Joy, G. Steele, and G. Bracha. *The Java Language Specification* (second edition). Boston: Addison-Wesley. 2000.

[4] Griswold, R. E. and M. T. Griswold. The Icon Programming Language (third edition). San Jose, Calif.: Peer-to-Peer Communications. 1997. www.cs.arizona.edu/icon/.

[5] Hanson, D. R. and M. Raghavachari. A machine-independent debugger. *Software—Practice and Experience* **26** (11), 1277–99. Nov. 1996. www.research.microsoft.com/~drh/pubs/cdb.pdf.

[6] Lewis, J. R., M. B. Shields, E. Meijer, and J. Launchbury. Implicit parameters: dynamic scoping with static types. *Conference Record of the 27th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Boston, 108–18. Jan., 2000. www.acm.org/pubs/articles/proceedings/plan/325694/p108-lewis/p108-lewis.pdf.

[7] Lindholm, T. and F. Yellin. *The Java Virtual Machine Specification* (second edition). Palo Alto, Calif.: Addison-Wesley. 1999.

[8] Nelson, G. *Systems Programming with Modula-3*. Englewood Cliffs, N.J.: Prentice-Hall. 1991. www.research.compaq.com/SRC/m3defn/html/m3.html.

[9] Proebsting, T. A. and G. M. Townsend. A new implementation of the Icon language. *Software—Practice and Experience* **30** (8), 925–72. July 2000. www.cs.arizona.edu/icon/jcon/impl.pdf.

[10] Richter, J. *Advanced Windows* (third edition). Redmond, Wash.: Microsoft Press. 1997.

[11] Scott, M. L. *Programming Language Pragmatics*. San Francisco: Morgan Kaufmann. 2000

APPENDIX A: SYNTAX SPECIFICATION

statement:

  `set` *def* { **`,`** *def* } **`in`** *statement*

  `use` *ref* { **`,`** *ref* } **`in`** *statement*

def: *identifier* **`:`** *type* **`=`** *expression*

ref: *identifier* **`:`** *type*

APPENDIX B: C++ MACROS

Listed below are macros that use the simple implementation technique to provide dynamic variables of pointers to class types in C++. Typical usage is:

`BEGIN_SET SET(`id_1, e_1**`); …; SET(`**id_n, e_n**`);`** S **`END_SET;`**

`BEGIN_USE USE(`id_1, T_1**`); …; USE(`**id_m, T_m**`);`** S **`END_USE;`**

Where the T_k are pointers to class types; dynamic casts are used to test subtypes. In **`USE(`***id*, *T***`)`**, if *id* is not found, *id* is set to 0. In the code below, **`Atom::New`** internalizes strings at program startup; i.e., it returns a pointer to the unique copy of its argument string. This code is available at: ftp://ftp.research.microsoft.com/Users/drh/dynamic.h and ftp://ftp.research.microsoft.com/Users/drh/dynamic.cpp.

```
class dVariable {
public:
  const char *name;
  void *value;
  class dVariable *link;
  dVariable(const char *name, void *value, class dVariable *link) :
    name(name), value(value), link(link) {}
};

class dEnvironment {
private:
  class dEnvironment *prev;
public:
  static class dEnvironment *current;
  class dVariable *vars;
  dEnvironment() : vars(current->vars), prev(current) { current = this; }
  ~dEnvironment() { current = prev; }
};

#define BEGIN_SET do { class dEnvironment _dEnv;

#define SET(id,e) \
  static const char *_name_##id = Atom::New(#id); \
  class dVariable _dvar_##id(_name_##id,e,dEnvironment::current->vars); \
  dEnvironment::current->vars = &_dvar_##id

#define END_SET } while (0)

#define BEGIN_USE do {

#define USE(id,T) \
  T id = 0; \
  do { \
    const char *_name = Atom::New(#id); \
    class dVariable *_p = dEnvironment::current->vars; \
    for ( ; _p; _p = _p->link) \
      if (_p->name == _name \
      && (id = dynamic_cast<T>(static_cast<T>(_p->value)))) \
        break; \
  } while (0)

#define END_USE } while (0)
```

This implementation is slightly different than the one described in Section 4.1. As depicted in Figure 2 below, `current` points to a `dEnvironment`, which has `prev` and `vars` fields. The `prev` field holds the previous value of `current`, and the `vars` field points to the list of `dVariable` instances. `current` is initialized to 0. The `BEGIN_SET` macro pushes a new `dEnvironment` instance onto the list headed by `current`. The `SET` macro declares a local variable for *id*, and pushes a `dVariable` onto the `current->vars` list. Using `dEnvironment`s makes it possible to remove all of the `dVariable`s created in `BEGIN_SET` … `END_SET` with the single assignment `current = prev` in the `dEnvironment` destructor.

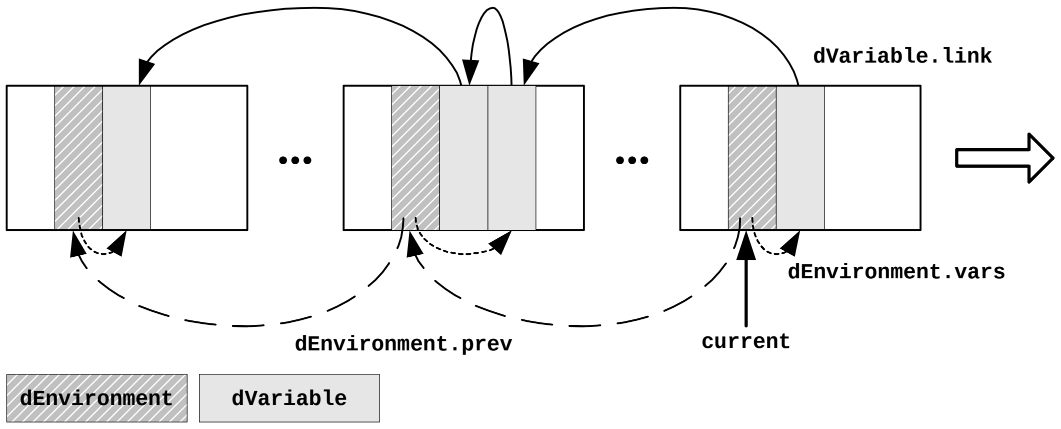

Figure 2. Shadow stack of `dEnvironment` and `dVariable` structures.

APPENDIX C: HASH-TABLE IMPLEMENTATION

The hash-table implementation described at the end of Section 4.1 involves only a few changes to the simple implementation. A single, per-thread global holds the hash table:

```
struct dVariable *dVars[HASHSIZE];
```

The `set` statement links new `dVariable` instances onto the appropriate hash chain. *H*("*id*") denotes the hash value for *id*, and the hashing and modulus shown below are computed at compile time. The `dVariable`s are unlinked after executing *S*.

```
{
   $T_1$  $id_1$ = $e_1$;
   struct dVariable dVar_$id_1$;
   dVar_$id_1$.name = "$id_1$";
   dVar_$id_1$.type = typeof ($id_1$);
   dVar_$id_1$.address = &$id_1$;
   dVar_$id_1$.link = dVars[$H$("$id_1$") mod HASHSIZE];
   dVars[$H$("$id_1$") mod HASHSIZE] = &dVar_$id_1$;
   …
   $T_n$  $id_n$ = $e_n$;
   struct dVariable dVar_$id_n$;
   dVar_$id_n$.name = "$id_n$";
   dVar_$id_n$.type = typeof ($id_n$);
   dVar_$id_n$.address = &$id_n$;
   dVar_$id_n$.link = dVars[$H$("$id_n$") mod HASHSIZE];
   dVars[$H$("$id_n$") mod HASHSIZE] = &dVar_$id_n$;
   $S$
   dVars[$H$("$id_n$") mod HASHSIZE] = dVar_$id_n$.link;
   …
   dVars[$H$("$id_1$") mod HASHSIZE] = dVar_$id_1$.link;
}
```

Some of the assignments following *S* can be omitted by unlinking only those **`dVariable`**s that were the first ones linked onto each hash chain. That is, the assignment for id_k is dead and can be omitted if it is known that the **`link`** field refers to id_j where $j < k$.

The **`use`** statement calls a revision of **`dSearch`** that searches just the appropriate hash chain:

```
{
  T1 *id1 = dSearch("id1", typeof (T1), H("id1") mod HASHSIZE);
  ...
  Tm *idm = dSearch("idm", typeof (Tm), H("idm") mod HASHSIZE);
  S
}
```

```
void *dSearch(const char *name, Type *type, int index) {
  struct dVariable *p;
  for (p = dVars[index]; p != 0; p = p->link)
    if (p->name == name && type is a subtype of p->type)
      return p->address;
  raise VariableNotFound;
}
```