Dynamic Variables

David R. Hanson and Todd A. Proebsting
Programming Language Systems Group
Microsoft Research
http://www.research.microsoft.com/pls/

One-Slide Summary

- Dynamic scope is useful; witness Perl, Tcl, …
- Two new statements for statically scoped languages
  - Instantiate variables with dynamic scope
  - Referencing them
- Intentionally minimalist design—use sparingly
- Good things
  - Customize execution environments, e.g., GUI libraries
  - Provide "thread-local" variables
- Simple implementation
- Better implementation

Static Scope is Pervasive?

- "All" modern languages use static scope
  - Efficient implementation
  - Compile-time error detection error detection
  - What you see is what you get—makes programs easy to understand
- "Older" languages use dynamic scope
  - Lisp, SNOBOL4, APL, ...
- But wait—some "newer" languages use dynamic scope
  - Shell environment variables
  - PostScript
  - TeX
  - Perl; on a per-variable basis
  - Tcl; via **`upvar`**

Dynamic Scope is Useful

- Customizable “environment”
  - GUI packages; window/widget attributes
  - Retargetable compilers; target specs, downstream clients
  - Component-based programming
- Without dynamic scope
  - Zillions of parameters (maybe with defaults)
  - Pointer to an “environment” with zillions of fields, methods
  - Global variables (!)
  - Amounts to implementing dynamic scope

Example: Printing Numbers

```
void print(int n) {
  char buf[43], *p = buf + sizeof buf;
  unsigned m = n;
  if (n < 0) m = -m;
  do
          *--p = m%10 + '0';
  while ((n /= 10) > 0)
  if (n < 0)
          *--p = '-';
  for ( ; p < buf + sizeof buf; p++)
          putc(*p, stdout);
}
```

- What about base, field width, emitter, file pointer?

```
void print(int n, int base, int width,
         void (*emitter)(char c)) { … }

void print(int n, struct { … } *env) { … }
```

An Old (and Persistent) Problem

- Ad hoc solutions
  - Type-unsafe
  - Inefficient
  - Problem-specific
  - Doesn't scale
  - Lack formal specs.
- Dynamic scope
  - Simple mechanism
  - Type-safe, amenable to formal specs.
  - Easy to distinguish lexically
  - Lewis et. al, POPL'2000: implicit parameters in Haskell

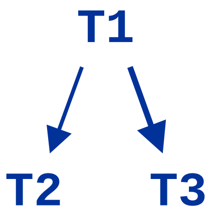
Design

- Two statements
  1. Instantiate a dynamically scoped variable—“dynamic variable”
  2. Bind a local variable to a dynamic variable
- **set** statement

  set *id* **:** *T* **=** *e* **in** *S*
  - Create *id* and initialize it to *e*, a subtype of *T* or assignable
  - *id* dies when *S* terminates
- **use** statement

  use *id* **:** *T* **in** *S*
  - Binds *id* to the most recently created dynamic variable *V*
  - *V* == *id* and type of *V* is a subtype of *T*

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



  - Static scope of *id* is restricted to *S*

Abbreviations

- Multiple declarations/bindings can appear in set/use

set *id1* **:** *T1* **=** *e1*, … , *idn* **:** *Tn* **=** *en* **in** *S*

set *id1* **:** *T1* **=** *e1* **in**
…
set *idn* **:** *Tn* **=** *en* **in** *S*

use *id* **:** *T1*, … , *idm* **:** *Tm* **in** *S*

use *id1* **:** *T1* **in**
…
use *idm* **:** *Tm* **in** *S*

Example Revisited (sans error-checking)

```
void print(int n) {
    char buf[8*sizeof n + 1], *p = buf + sizeof buf;
    unsigned m = n;
    use base: int in {
            if (n < 0) m = -m;
            do
                    *--p = ”01234567890abc…xyz”[m%base];
            while ((n /= base) > 0)
            if (n < 0)
                    *--p = ’-’;
    }
    use width: int, emitter: void (*)(char) in {
            int len = (buf + sizeof buf) – p;
            for ( ; width > len; len++)
                    (*emitter)(’ ’);
            for ( ; p < buf + sizeof buf; p++)
                    (*emitter)(*p);
    }
  }
```

Compiling Loops and Switches in lcc

- Pass loop and switch handles to *every* parsing function

```
void statement(int loop, Swtch *switchp) {
    …
    forstmt(newlabel(), switchp);
    …
    switchstmt(loop, newswitch());
    …
}
```

  - while/for/do statements produce **loop**
  - switch statements produce **switchp**
  - continue statements consume **loop**
  - case/default statements consume **switchp**
- Use **set** in producers, **use** in consumers: Zap clutter

Thread-Local Variables

- Some languages provide “thread-local” variables
  - Global scope—per thread
  - Lifetimes associated with thread lifetimes
  - Microsoft Visual C++ uses Windows “thread-local storage”
  - Doesn’t work in libraries loaded at runtime (!)
- Dynamic variables are automatically thread-local
  - Set them in thread’s initial function
  - Use them in other functions
- Unappreciated benefit of <u>all</u> dynamic scope mechanisms

Implementation Techniques

- Simple implementation
  - Reasonably efficient: no allocation, linear search
  - Easy to get correct
- **set** *id* **:** *T* **=** *e* **in** *S*

  push { address of *id* : *T* } onto a per-thread stack
  S
  pop
- **use** *id* **:** *T* **in** *S*

  search stack for *id* : *T* upon statement entry only
  S (access id by indirection)
- 90% solution: C++ macros for pointers to class types
- "Novel" implementation
  - For languages with exception handling
  - Builds on existing compiler infrastructure for exceptions

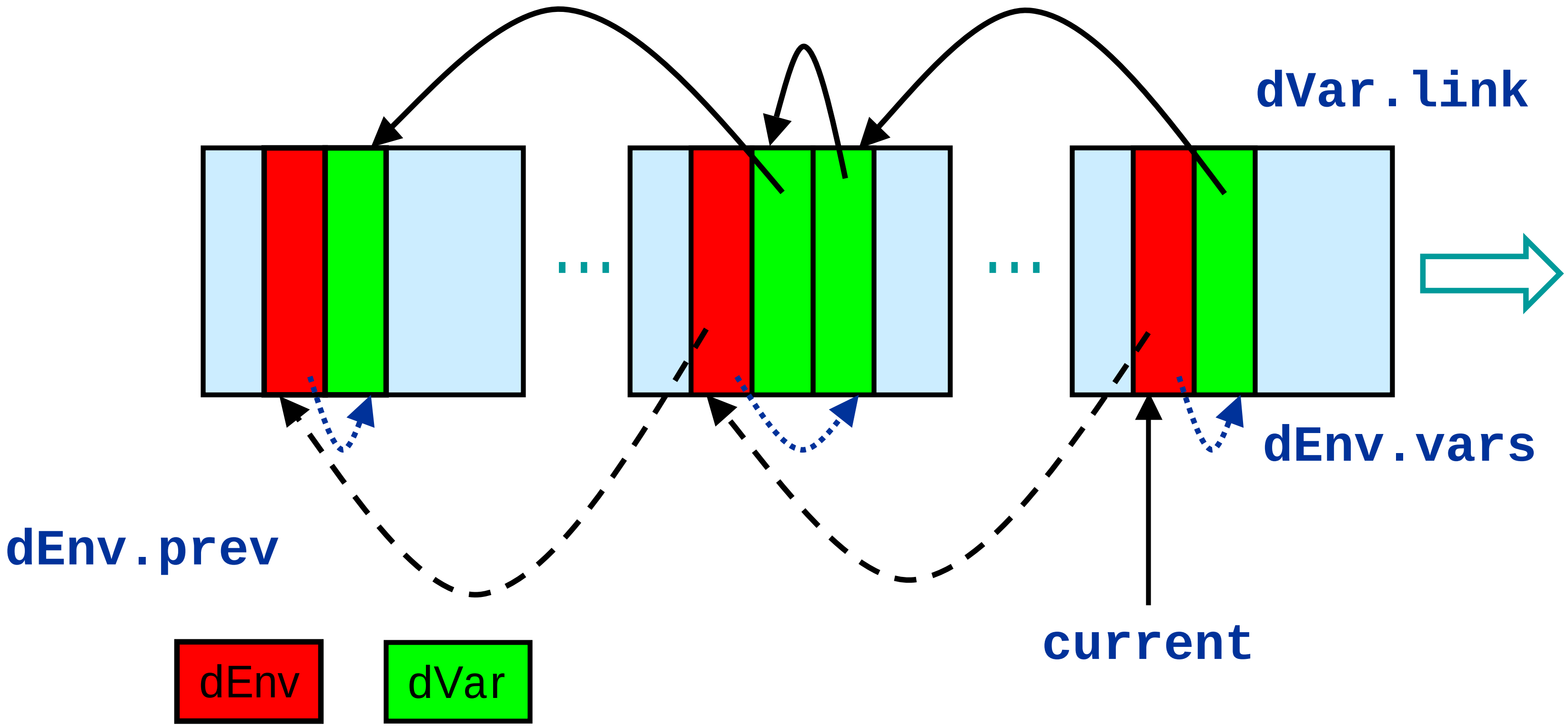
Simple Implementation—Set

set *id1* **:** *T1* **=** *e1***,** …**,** *idn* **:** *Tn* **=** *en* **in** *S*

```
current = dEnv { prev = current,
                 vars = current->vars };
T1 id1 = e1;
current->vars = dVar { name = "id1", type = T1,
    address = &id1, link = current->vars };
…
Tn idn = en;
current->vars = dVar { name = "idn", type = Tn,
    address = &idn, link = current->vars };
S
current = current->prev;
```

- *No* runtime allocation—dEnv's and dVar's are locals

Shadow Stack



Simple Implementation—Use

use *id1* **:** *T1***, …,** *idm* **:** *Tm* **in** *S*

```
T1 *id1 = dSearch("id1", T1)
…
Tm *idm = dSearch("idm", Tm)
S
void *dSearch(char *name, Type *type) {
    dVar *p = current->vars;
    for ( ; p != 0; p = p->link)
          if (p->name == name
          && type is a subtype of p->type)
                return p->address;
    RuntimeError();
}
```

- Names are "internalized"—one copy of each name

“Novel” Implementation

- “Standard” exception-handling table entries (e.g. Java)

`void *from`	start of PC range
`void *to`	end of PC range
`void *handler`	address of exception handler
`Type *type`	exception type

- Extend tables with two entries for set statements

`void *from`	start of set statement
`void *to`	end of set statement
`char *name`	identifier name
`Type *type`	identifier type
`int offset`	frame offset

- Like the **`try`** statement, **`set`** has no time overhead
- **`use`** walks stack, interprets tables

Novel Implementation—Set

set *id1* **:** *T1* **=** *e1*, …, *idn* **:** *Tn* **=** *en* **in** *S*

```
              T1 id1 = e1;
start1:       T2 id2 = e2;
      …
startn-1:     Tn idn = en;
startn:       S
end:
```

from	to	type	name	offset
start1	end	*T1*	"*id1*"	*offset1*
start2	end	*T2*	"*id2*"	*offset2*
…				
startn	end	*Tn*	"*idn*"	*offsetn*

Novel Implementation—Use

use *id1* **:** *T1***, …,** *idm* **:** *Tm* **in** *S*

```
T1 *id1 = dLookup("id1", T1)
…
Tm *idm = dLookup("idm", Tm)
S
void *dLookup(char *name, Type *type) {
    for each stack frame f
        for each table entry t
            if (pc >= t.from && pc < t.to
            && t.name == name
            && type is a subtype of t.type)
                return f + t.offset;
    RuntimeError();
}
```

Enhancements

- Design is intentionally minimalist
- Dynamic variables are best used sparingly—like exceptions, but…
- Avoid superfluous declarations
  - Abbreviate *T id*; …; **set** *id* : *T* = *id* **in** *S* by **set** *id* **in** *S*
- Avoid name conflicts
  - **use** *id* : *T* **as** *id'* **in** *S*
- Handle missing variables, set defaults
  - Raise exception on missing variables
  - Boolean **isdynamic(***id*, *T* **)**
  - **use** *id* : *T* = *default* **in** *S*
    if "**=** *default*" is omitted, missing *id* is an error/exception (interesting implementation issues)

Acceptance?

- Exceptions are a <u>control</u> construct with dynamic scope
  - Avoids clutter
  - Helps build reliable and adaptable software
  - Exception handling is now widely accepted
- Dynamic variables are a <u>data</u> construct with dynamic scope
  - Avoids clutter
  - Easy/efficient addition to languages with exception handling

- What happens next?
  - Experimental implementation (perhaps in C#)
  - Proposed for addition to C# (not likely…)