Dynamic Variables

David R. Hanson and Todd A. Proebsting

Programming Language Systems Group
Microsoft Research

http://www.research.microsoft.com/pls/

One-Slide Summary

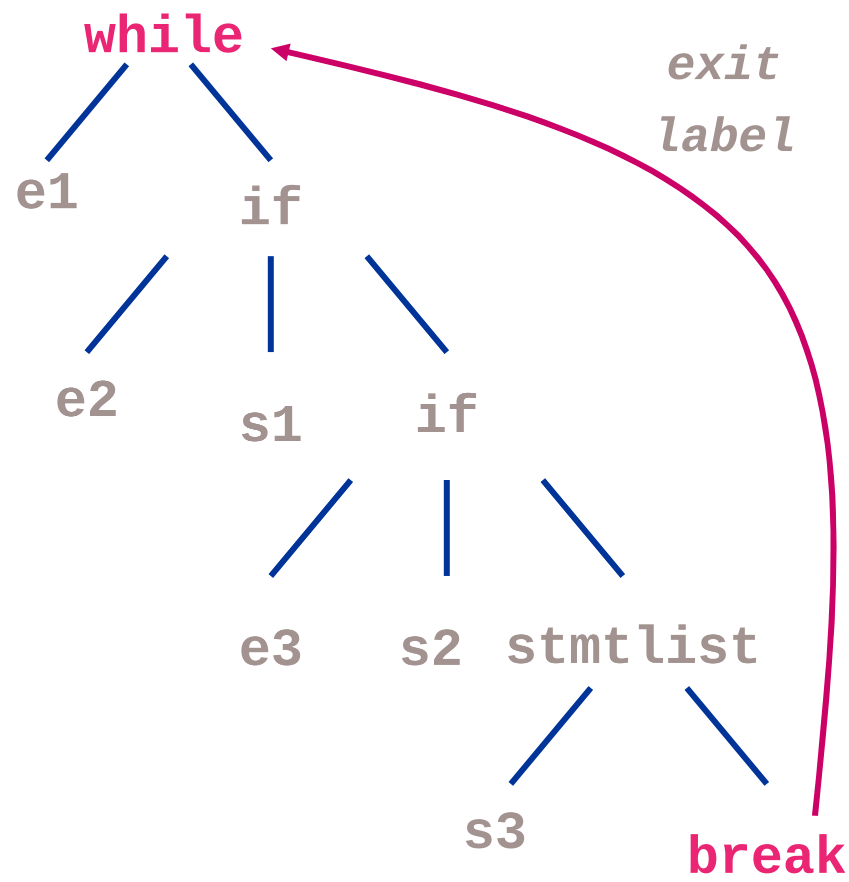
- Dynamic scope is useful
  - Customize execution environments, e.g., GUI libraries
  - Provide "thread-local" variables
- Two new statements for statically scoped languages:
  - Instantiate variables with dynamic scope
  - Reference them
- Intentionally minimalist design—use sparingly
- Implementations
  - Simple
  - Better

Example: Compiling Loops and Switches in lcc

- Compile

```
while (e1)
    if (e2)
        s1
    else if (e2)
        s2
    else {
        s3;
        break;
    }
```

- Recursive-descent tree-walk



Example: Compiling Loops and Switches in lcc

- Recursively 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`**
  - break and continue statements consume **`loop`**
  - case/default statements consume **`switchp`**

- These arguments are almost never used!

An Old (and Persistent) Problem

- Ad hoc, problem-specific solutions
  (E.g., global variables, environment parameters)
  - Inefficient
  - Don't scale
  - Lack formal specs.

- Dynamic scope solution:
  - Simple mechanism
  - Type-safe, amenable to formal specs.
  - Easy to distinguish lexically
  - Treated rigorously (and supported enthusiastically) in Lewis et. al, POPL'2000: "implicit parameters" in Haskell

Our Simple Design: Instantiate and Bind

- Instantiate a "dynamic variable" with **set** statement

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

  - Create *id* with type *T*, and initialize it to *e*
  - *id* has dynamic scope within S
  - *id* dies when *S* terminates

- Bind to a dynamic variable with **use** statement

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

  - Binds **local** *id* to the most recently created dynamic variable *V* *such that*
    - ***V* == *id***
    - **type of *V* is a subtype of *T***
  - Static scope of *id* is restricted to *S*

Simple Binding Semantics

```
A() {
   set x : T3 = … in { … B() … }
}

B() {
   set x : T2 = … in { C() }
}

C() {
   use x : T1 in { … }
}
```

Example 1

$$\begin{tikzcd} & T1 \arrow[dl] \arrow[dr] & \\ T2 & & T3 \end{tikzcd}$$

Example 2

$$\begin{tikzcd} T2 & T1 \arrow[d] \\ & T3 \end{tikzcd}$$

- (Sub-)Type discrimination vital to component-based applications

Example Revisited: Compiling Loops/Switches in lcc

- Use **set** in producers, **use** in consumers: Zap clutter

```
void statement() {  // without parameters!
    …
    forstmt();
    …
    breakstmt();
    …
}
void forstmt() {
    set loop: int = newlabel() in { … statement() … }
}
void breakstmt() {
    use loop: int in { … }
}
```

- Usable C++ implementation available; see paper

Our Implementation Techniques

- Simple implementation (C++ in paper)
  - Reasonably efficient: no allocation, linear search
  - Easy to get correct

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

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

- **use** *id* **:** *T* **in** *S*

  search stack for *id* : *T* upon statement <u>entry only</u>
  S (access *id* by indirection)

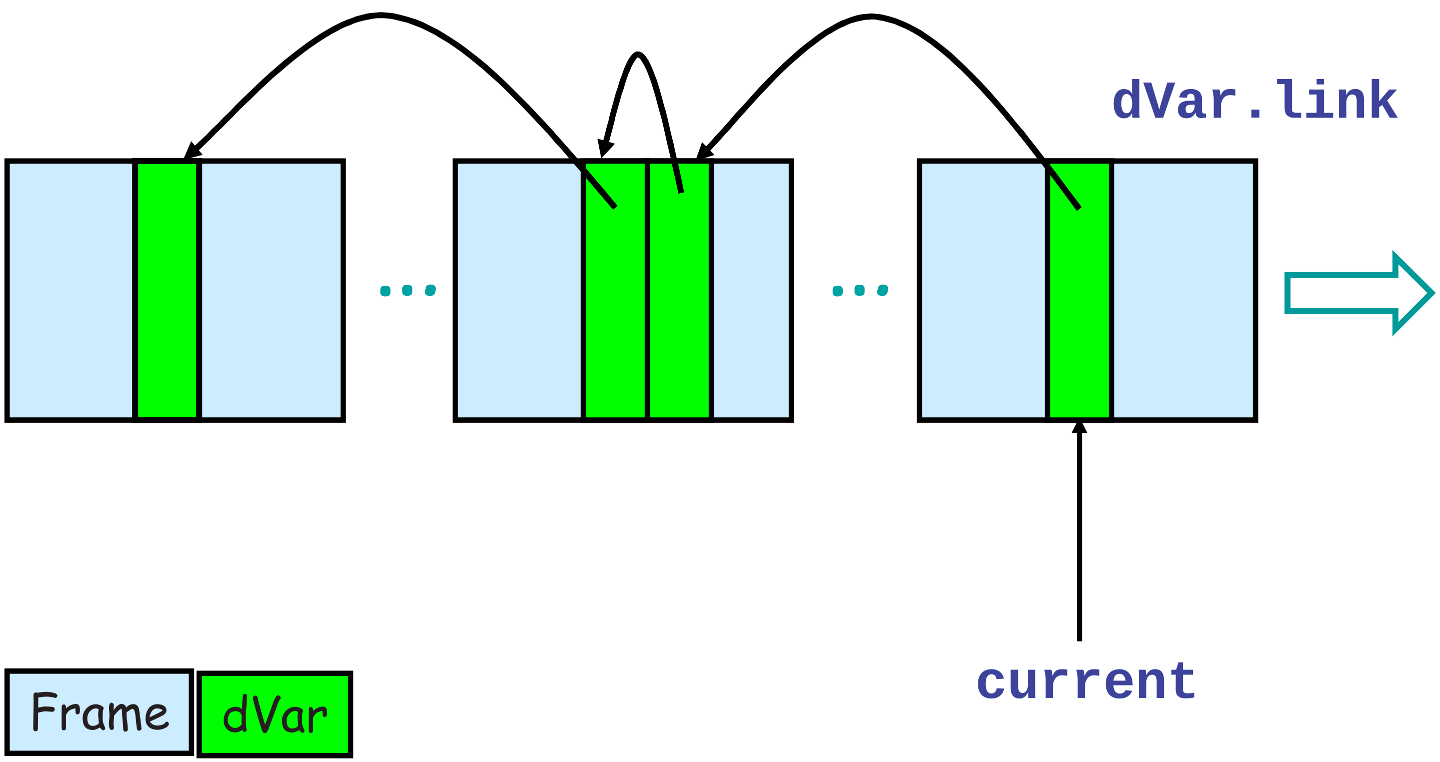
Simple Implementation—Set

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

```
T id = e;
current = & dVar {         // “push” dynamic descriptor
    name = “id”,           // name for dynamic lookup
    type = T,              // type for dynamic lookup
    address = &id,         // address for “use” binding
    link = current         // link to previous descriptor
};
S
current = current->link;// “pop” dynamic scope
```

- Linked list with no runtime allocation:
  dynamic variable descriptors are locals

Shadow Stack Via Linked List



Simple Implementation—Use

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

T ***** *idptr* **= dSearch("***id***",** *T***)**
S *// access id indirectly via idptr within S*

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

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

“Novel” Implementation

- Builds on existing compiler infrastructure for exceptions
- “Standard” exception-handling table entries (e.g. Java)

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

- Adapt tables with two entries for set statements

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

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

Novel Implementation—Set

set *id* **:** *T* **=** *e* **in** *S*   *translates to*

code:

```
        T id = e;
start:  S
end:
```

Per function table:

from	**to**	**type**	**name**	**offset**
start	end	*T*	*"id"*	*id's offset*

Note: Table has one entry per 'set' dynamic variable

Novel Implementation—Use

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

```
T *idptr = dLookup("id", T)
S  // access id indirectly via idptr within 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;
    raise VariableNotFound;
}
```

Lexical scoping + set/use = no "funarg" problem

- funarg problem solved with
  - lexical scoping
  - controlled dynamic scoping: distinguish binding from using variable

- use *inside* lambda:
  binds to dynamic variable visible from caller

```
void f() {
    fn = lambda() {
        use free : T in {
            … free …
        }
    }
    return fn;
}
```

- use *outside* lambda:
  binds to dynamic variable visible from parent

```
void f() {
    use free : T in {
        fn = lambda() {
            … free …
        }
    }
    return fn;
}
```

Dynamic Variables: 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

Enhancement?: Default values

- Design is intentionally minimalist
- Dynamic variables are best used sparingly—like exceptions, but…

- Handling missing variables…
  - Raise exception on missing variables
  - Boolean **`isdynamic(`***id*, *T* **`)`** ala **`instanceof(…)`**
  - Mechanism for default values of missing variables?

Acceptance?

- Exceptions are a control construct with dynamic scope
  - Avoids clutter
  - Helps build reliable and adaptable software
  - Exception handling is now widely accepted
- Dynamic variables are a data construct with dynamic scope
  - Avoids clutter
  - Easy/efficient addition to languages with exception handling
- What happens next?
  - Experimental implementation in C# (someday)
  - Proposed for addition to C# (not likely…)