Cecil Language Introduction

Dave Grove and Jeff Dean

CSE 501 Winter 1995

Cecil Concepts

Pure language model

- User defined control structures
- User defined primitive operations (eg. arithmetic)
- All calls are potentially a message send

Dynamic Typing

Garbage Collection

Multi-methods

Closures

Memory Safe

Examples

```
method factorial(n) {
    if(n = 0,
        {1},
        {n * factorial(n - 1)}) }

factorial(7)
```



```
method -(x@num, y@num) { x.asFloat + y.asFloat }

method -(x@int, y@int) { ... code to add 2 integers ...}
method asFloat(x@int) { ... convert x to a float ... }

method -(x@float, y@float) { ... code to add 2 floats ...}
method asFloat(x@float) { x }
```

What happens when we type...

```
3 -5

4.5 -3.7

4 -3.2

"hi" - 3.5
```

```
abstract object bool;
concrete object true isa bool;
concrete object false isa bool;

method if(t@bool, tc) { if(t, tc, {}) }
method if(@true, tc, fc) { eval(tc) }
method if(@false, tc, fc) { eval(fc) }
```


What is a program?

```
program ::= { decl | stmt | expr }
```

When a program is run:

- Initialize each declaration in order
- Evaluate stmts & exprs in order, returning the result of the last expression in the program

Declarations

4 Main kinds of declarations

- variable
- object
- methods
- fields

See the language reference manual for others

Var Decls

```
var_decl ::= "let" ["var"] name [type_decl] ":=" expr ";"

let x:int := 3;
let var y:string := "this is an example";
let z := 3.14159;
y := "another_string";
```

Variables must be initialized

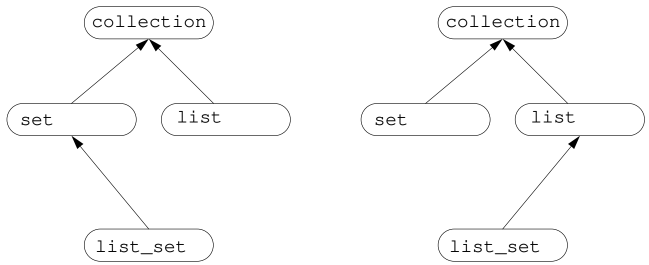
Only variables declared with "var" are mutable

Standard scoping rules.

Object_Decls

Cecil separates subtyping from inheritance

```
abstract object collection;
abstract object set isa collection;
template object list isa collection;
template object list_set inherits list subtypes set;
```



Type Graph Inheritance Graph

Method Decls

```
method_decl ::= "method" msg_name "(" {formal} ")"
                "{" body "}"

formal ::= name | name@obj


method factorial(t@int) { }
method factorial(t) { }

method if(@true, tc) { ..... }
```


Method Bodies

```
body ::= {stmt ";"}

stmt ::= var_decl
      |   assignment
      |   expr


assignment ::= name ":=" expr

x := factorial(3 + 5);
y := 3.5;
```


Exprs

```
expr ::= literal | simple_expr | message

literal ::= integer | float | character | string

simple_expr ::= name | array_expr
               | closure_expr | object_expr

array_expr: [3, 4, "hi", 4.0]

closure_expr: &(x,y) { x + 5 }
              &() { factorial(10) }
              { factorial(10) }

object_expr: concrete object isa list_set
```

Our current implementation only supports LIFO closures!

(no upward funargs)

Message Exprs

```
length(c);
c.length;

x + y
x ! y

c.do(y)
do(c, y)

x.foo(1,2,3,4)
foo(x,1,2,3,4)
```


Closures

```
method example(y) {
    let var x := 5;
    if(y = 3, {
      x := x + 3;
    }, {
      ^15
    })
    x }

example(3) + example(4)
```


Field Decls

Like an instance variable

Allocates storage & defines 2 methods

- get accessor, to get the value of the field
- set accessor, to set the value of the field

```
object foo;
  var field first(@:foo):int := 3;
  field second(@:foo):float;
  field third(t@:foo) := new_array(t.first);

let x := concrete object isa foo { second := 2.56 }

x.first := 12; -- sugar for set_first(x, 12)
x.second;
x.third -- initializer evaluated here!
```


Type Declarations

Static type declarations are optional in Cecil

- Can omit type declarations initially, while still getting code to work, and then add them later, to get static checking
- Type checker for these optional declarations still in progress

Even without the type checker, type declarations are useful: serve to document the code

Parameterized Types

Useful to reuse code that differs only in the types of elements they will contain

- Heavily used in standard collection hierarchy

Examples:

```
abstract object collection[T];
abstract object table[Key, Value] isa collection[Value];
abstract object indexed[T] isa table[int,T];

template object array[T] isa indexed[T];

method new_array[T]():array[T] { concrete object isa array[T] }
method fetch(a@:array[`T], index@:int, if_absent:&():T):T {...}

let x:array[int] := new_array[int]();
x.fetch(7, { error("element not found") })

let x:table[rtl_symbol, constant_info] :=
    new_hash_table[rtl_symbol, constant_info]();
```


Towers of Hanoi - A Bigger Example

First define some data structures:

```
template object towers;
field stacks(@:towers):m_indexed[stack[disk]];
var field moves_done(@:towers):int := 0;

template object disk;
field id(@:disk):int;
method print_string(d@:disk):string { d.id.print_string }
```

And a routine to set them up:

```
method create_towers(size:int):towers {
    let stack0:stack[disk] := new_stack[disk]();
    let stack1:stack[disk] := new_stack[disk]();
    let stack2:stack[disk] := new_stack[disk]();
    do(size, &(i:int){
  push(stack0, object isa disk { id := size - i });
    });
    object isa towers { stacks := [stack0, stack1, stack2]
} }
```


Towers (cont)

```
method move_stack(t@:towers, from:int, to:int,
count:int):void {
    if(count = 0, { ^ });
    let other:int := 3 - from - to;
    move_stack(t, from, other, count - 1);
    move_disk(t, from, to);
    move_stack(t, other, to, count - 1);
};

method move_disk(t@:towers, from:int, to:int):void {
    let d:disk := pop(t.stacks!from);
    if((t.stacks!to).non_empty &
             { (t.stacks!to).top.id < d.id },
       { error("pushing a disk onto a smaller disk"); });
    push(t.stacks!to, d);
    t.moves_done := t.moves_done + 1;
}
```


Towers (cont)

Now define a starting routine:

```
method solve_towers(size:int):void {
    ("** Starting towers with " || size.print_string ||
     " disks...\n").print;

    let time:int := time({
        let t:towers := create_towers(size);
        move_stack(t, 0, 1, size);
    });
    "done.  ".print;
    ("time: " || time.print_string || " ms.").print_line;
}
```

And a top-level expression:

```
solve_towers(argv.fetch(1, { 10 }))
```

Cecil Stdlib

Defines a whole bunch of useful things:

- Primitive data types & operations
- Control Structures
- ADT's
  - lists, sets, arrays, streams, hash_tables, etc. etc.

Check out the Stdlib reference manual.

(Available from internal documentation link from Cecil project home page)

An Interesting Example

Dead Assignment Elimination in the Vortex compiler

```
template object dead_assign_info isa analysis_info;
  var field seen(@:dead_assign_info):symbol_bit_set :=
           new_symbol_bit_set();

method dead_assign_elim(g@:CFG_context):void {
  traverse(g, reverse_traversal, iterative,
           new_dead_assign_info(g.scope),
          &(n:RTL, info:dead_assign_info) {
         if(n.can_elim(info),
            { new_delete_CFG_straight(info) },
            { info.change_info(n);
              new_continue_single_info(info) }) });
}
```



```
method can_elim(name@:rtl_symbol,
                s@:dead_assign_info):bool {
    not(s.seen.includes(name) }

-- default this to false
method can_elim(@:RTL, @:dead_assign_info):bool { false }

method can_elim(t@:rtl_move, s@:dead_assign_info):bool {
    t.result.can_elim(s) }

method change_info(s@:dead_assign_info, t@:RTL):void {
    t.defs.do(&(n:rtl_symbol){
       s.seen.remove(n, { (-- ok if not there --) });
    });
    t.uses.do(&(n:rtl_symbol){
      if_false(n.is_literal, {
         s.seen.add(n);
      });
    });
}
```