Compilation and bytecode execution

Jay Foad

DYALOG

Sicily 2015

Compiler

- Introduction / recap
- New features in 14.1
- Results
- Future work

Recap

- Compiler ***compiles*** defined functions (dfn or tradfn) into ***bytecode***
- Bytecode executes more efficiently
- Reduces interpreter overhead
- Speeds up "the invisible glue between the tokens" –Nick Nickolov
- Can speed up your code IF it's working on scalars or small arrays

Recap: limitations

- Fundamental restriction: compiler must be able to resolve names
- ... or at least know their nameclass

Recap: limitations

- Fundamental restriction: [redacted] must be able to resolve name[redacted]
- ... or at least know their [redacted] syntactic category
  - array (or niladic function)
  - function
  - monadic operator
  - dyadic operator

Recap: limitations

- Fundamental restriction: compiler must be able to resolve names

```
f←{ t ← 1.8 × ⍵ ⋄ 32 + t }
```

Recap: limitations

- Fundamental restriction: compiler must be able to resolve names

```
f←{ t ← 1.8 × ω ⋄ 32 + t }
```

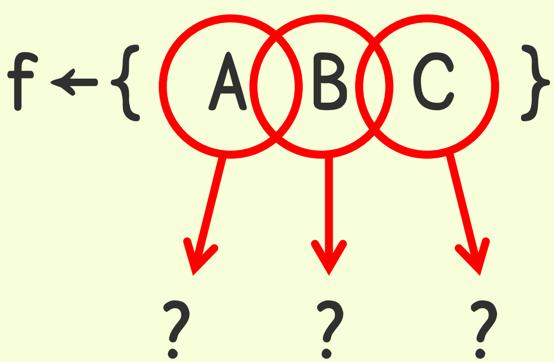
Recap: limitations

- Fundamental restriction: compiler must be able to resolve names

```
f←{ A B C }
```

Recap: limitations

- Fundamental restriction: compiler must be able to resolve names



Recap: limitations

```
f←{
    a ← ⍺
    b ← ⍎⍵
    a
}
```

Recap: limitations

```
f←{
?   a ← ⍺
    b ← ⍎⍵
    a
}
```

Recap: UI

- Compiler is controlled by an I-beam:

```
cc ← 400⌶  ⍝ compiler control
```

Compiler in 14.0

```
cc ← 400⌶
1 cc'foo' ⍝ is foo compiled?
2 cc'foo' ⍝ compile foo
3 cc'foo' ⍝ uncompile foo
```

Compiler in 14.1

```
cc ← 400⌶
1 cc'foo' ⍝ is foo compiled?
2 cc'foo' ⍝ compile foo
3 cc'foo' ⍝ uncompile foo
4 cc'foo' ⍝ show bytecode
# cc'foo' ⍝ compile foo with
          ⍝   global names
```

Compiler in 14.1

```
cc ← 400⌶
iscompiled ← 1 ∘ cc
   compile  ← 2 ∘ cc
uncompile  ← 3 ∘ cc
 bytecode  ← 4 ∘ cc
```

New features in 14.1

- Global names
- Bytecode display
- Control structures and tradfns

Global names

- In 14.0, any use of a non-local name caused an error

```
      bar←{ω*2}
      foo←{bar 1+ω}
      compile'foo'
16 0 0 Undefined name: bar
```

- In 14.1 a *callback* mechanism lets you overcome this

Global names

- TL;DR

```
        quadNC ← ⎕NC
        quadAT ← ⎕AT
        # cc'foo'
⍝ (success)
```

Global names

- What happened there?
- Left argument of I-beam is a ***namespace***
- Namespace contains well-known named ***callback functions***
- Compiler uses them to resolve unknown (non-local) names

Global names

Callbacks `quadNC` and `quadAT` are called with a single enclosed name:

```
quadNC ⊂'bar'
quadAT ⊂'bar'
```

For `quadAT`, only the first item of the result (valency) is significant

Why use callbacks?

- To ***decouple*** the compile-time and run-time environments
  - e.g. when code is loaded dynamically
  - e.g. when you have generated constants
- (To work around some problems)
- If you don't care, use the defaults

Why use callbacks?

```
quadNC←{
    ⍵∊FuncNames:3.2  ⍝ dfn
    0
}
```

Why use callbacks?

```
quadNC←{
      'C'≡⊃⊃⍵:2.1  ⍝ Constant
      'F'≡⊃⊃⍵:3.1  ⍝ Function
      0
}
```

Why use callbacks?

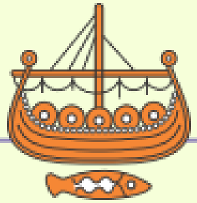
```
quadNC←{
    ⍵≡⊂'badfunc':0
    ⍵≡⊂'reallybadfunc':0
    ⎕NC ⍵
}
```

Why use callbacks?

```
quadNC←⎕NC
```

Assumptions

- The nameclass of a global name is recorded in the bytecode as an ***assumption***
- Assumptions are checked at run time



Assumptions

- What if the assumption fails?

```
      quadNC←{3.1} ⋄ quadAT←{(1 0 0) 0 0 0}
      bar←99
      foo←{bar ⍵}
      # cc'foo'    ⍝ compilation succeeds!
      foo 3
SYNTAX ERROR: Nameclass of non-local name has
   changed since compilation
```

Global constants

- Callback function `getValue` can return the ***value*** of a global constant, enclosed
- ... else `⍬`
- You are promising the compiler that the value won't change
- This assumption is not checked!

Global constants

```
quadNC←{
    'C'≡⊃⊃ω:2.1  ⍝ Constant
    0
}
getValue←{
    'C'≡⊃ω:⊂⍎ω   ⍝ or ⊂⎕OR ω
    ⍬
}
```

Global constants

- Why isn't the assumption checked?
- Because of constant folding

```
C3←1 2 3    ⍝ constant
foo←{≢C3}
# cc'foo'
```

Bytecode

- `4(400⌶)` dumps the bytecode of a compiled function

Health warning:

- This is for interest and amusement only!
- The bytecode display can and will change at any time!

Bytecode

```
      f←{⍺+⍵}
      compile'f'
      bytecode'f'
Dump of bytecode for f:
  0000: 0000000F // version 15
  0001: 00000000 // localised system variables: none
  0002: 00000001 // 0 slots
  0003: 00000002 // 0 uslots
  0004: 00000224 eval 0x02 // +
  0005: 00003A12 tokoff 003A
  0006: 00000003 ret
```

Bytecode: slots

```
      g←{(1+⍵)÷(1-⍵)}
      compile'g'
      bytecode'g'
Dump of bytecode for g:
  0000: 0000000F // version 15
  0001: 00000000 // localised system variables: none
  0002: 00000201 // 2 slots
  0003: 00000002 // 0 uslots
  0005: 00003125 cpy Larg, lst[1]
  0006: 00000F46 cpy slot[0], Rarg
  0007: 00000324 eval 0x03 // -
  0009: 00003125 cpy Larg, lst[1]
  000A: 00000E45 mov Rarg, slot[0]
  000B: 00002E66 mov slot[1], Rslt
  000C: 00000224 eval 0x02 // +
  000E: 00006025 mov Larg, Rslt
  000F: 00002E45 mov Rarg, slot[1]
  0010: 00000524 eval 0x05 // ÷
  0012: 00000003 ret
```

Bytecode: recursion

```
      gcd←{⍵=0:⍺ ⋄ ⍵ ∇ ⍵|⍺}
      compile'gcd'
      bytecode'gcd'
Dump of bytecode for gcd:
  0000: 0000000F // version 15
  0001: 00000000 // localised system variables: none
  0002: 00000201 // 2 slots
  0003: 00000002 // 0 uslots
  0004: 00000E26 mov slot[0], Larg
  0006: 00004125 cpy Larg, Rarg
  0007: 00002E46 mov slot[1], Rarg
  0008: 00003145 cpy Rarg, lst[1]
  0009: 00001524 eval 0x15 // =
  000B: 00006045 mov Rarg, Rslt
  000C: 0000100F jumpfalse 0010
  000D: 00000F65 cpy Rslt, slot[0]
  000E: 00000003 ret
  0010: 00000F65 cpy Rslt, slot[0]
  0011: 00002F25 cpy Larg, slot[1]
  0012: 00006045 mov Rarg, Rslt
  0013: 00000A24 eval 0x0A // |
  0015: 00002E25 mov Larg, slot[1]
  0016: 00006045 mov Rarg, Rslt
  0017: 00000411 tailrecurse 0004
```

Bytecode

- Values are moved around in ***registers*** `Larg, Rarg, Rslt`
- Constants loaded from `lst[n]`
- Temporaries stored in `slot[n]`
- Functions executed with `eval`

Control structures

- In 14.0 the compiler (mostly) just targeted dfns
- In 14.1 both branch (→) and all normal control flow structures are supported
  - `→label` is special-cased
  - `→expression` is less efficient

Control structures

```
     ∇ n←loop n
[1]    :Repeat
[2]      n←n-1
[3]    :Until n=0
     ∇
```

```
0005: 00000E46 mov slot[0], Rarg
0006: 00000F25 cpy Larg, slot[0]
0007: 00005145 cpy Rarg, lst[2]
0008: 00000324 eval 0x03 // -
000B: 00000F66 cpy slot[0], Rslt
000C: 00006025 mov Larg, Rslt
000D: 00007145 cpy Rarg, lst[3]
000E: 00001524 eval 0x15 // =
0010: 00006045 mov Rarg, Rslt
0011: 0000060F jumpfalse 0006
0012: 00000F65 cpy Rslt, slot[0]
0013: 00000003 ret
```

Control structures

```
     ∇ n←osc n;cond;t
[1]    :Repeat
[2]        cond←2|n ⋄ :If cond
[3]            t←1+3×n
[4]        :Else
[5]            t←n÷2
[6]        :End ⋄ n←t
[7]    :Until n=1
     ∇
```

Factor of 2.5 speed-up

Coverage

On a large application with 64501 defined functions (0.1% dfns)

- < 1% in 14.0
- 59.15% in 14.1
- 79.05% with indexed assignment
- 83.47% with selective assignment

Coverage

The next top priorities (for this code base)

```
5428 Non-local assignment
4209 Dotted namespace reference
1003 Execute
876 Unsupported system function: ⎕WG
594 Unsupported system function: ⎕WS
576 Unsupported system function: ⎕NC
396 Unsupported keyword: :With
365 Unsupported keyword: :Trap
...
```

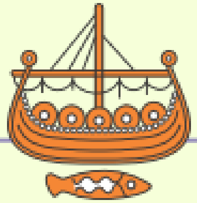
Coverage

- But this sample is biased!
- 99.9% tradfns
- For example, no use of right argument namelists:

```
     ∇ r←foo(x y z)
[1]    r←x+y×z
     ∇
```

Future work: coverage

- Indexed assignment
- Selective assignment
- Right argument namelist
- Better support for namespaces



Future work: performance

- Inline single-line dfns
- Better constant folding
- Better support for ⎕ML ⎕IO etc
- More idioms: 0=N| (for all N)
- Scalar loop fusion: A+B×C

Feedback

Please:

- Try it out
- Report failures
- Report successes
- Send us your code!