CONST-DOES>

M. Anton Ertl
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
anton@mips.complang.tuwien.ac.at
http://www.complang.tuwien.ac.at/anton/
Tel.: (+43-1) 58801 18515
Fax.: (+43-1) 58801 18598

Abstract

A frequent use of the `create`...`does>` construct is to provide some constants at definition time that are then used at execution time (a `constant`-style use). However, `create`...`does>` also supports a `value`-style use, where the data is not constant and can change at any time. This additional functionality inhibits optimization. This paper proposes `const-does>`, which can only be used to define `constant`-style words, and thus makes optimization possible.

This paper is written in the form of a proposal like those on http://www.complang.tuwien.ac.at/forth/ansforth/proposals.html.

1 Problem

Many uses of `create`...`does>` are just for shifting data from the `create` time to the execution time of the code after `does>`; i.e., after the word is fully defined, the data remains constant. A prototypical example of this use is the definition

```
: constant ( n "name" -- )
  create ,
does> ( -- n )
  @ ;

42 constant answer
```

Here, *n* is just shifted from `create` time to *name* execution time.

It would be nice if a native-code compiler could optimize a use of `answer` in the same way that it would optimize a use of `42`. However, this is not possible, because the compiler has to consider the following possibility:

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in: EuroForth 2000 Proceedings, pages 13-17

```
5 ' answer >body !
```

I.e., the data in a `create`...`does>`-defined word can change at almost any time. So at best a compiler can compile `answer` to the same code as `[ ' answer >body ] literal @`.

The effects of this difference on the resulting code depend on the context. E.g., consider the code `answer cells + @`: If the compiler could optimize `answer` to `42`, it could compile this sequence to one instruction on the MIPS architecture:

```
lw        v0,168(a0) ; 42 cells + @
```

Without this optimization, it needs at least five instructions:

```
lui       v1,...    ; [ ' answer >body ] literal
lw        v0,...(v1); @
sll       v0,v0,2   ; cells
addu      v0,v0,a0  ; +
lw        v0,0(v0)  ; @
```

2 Proposal

`CONST-DOES>` "const-does" core

Interpretation: Interpretation semantics for this word are undefined.

Compilation: (C: colon-sys1 – colon-sys2)

> Append the run-time semantics below to the current definition. Whether or not the current definition is rendered findable in the dictionary by the compilation of `const-does>` is implementation defined.

Run-time: (u1*x u2*r u1 u2 "name" R: nest-sys1 –)

> Create a word *name* with execution semantics given below. Return control to the calling definition specified by *nest-sys1*. The *u1* cells and *u2* floats can be interleaved in any order.

name execution: (... – ...)

> Perform initiation semantics below. Transfer control to the code right after the `const-does>`.

Initiation: (– u1*x u2*r R: nest-sys2)

> Save information *next-sys2* about the calling definition. After pushing the *u1* cells and *u2* floats, they are in the same order as they were at the start of the run-time semantics.

3 Typical use

```
: constant ( n "name" -- )
1 0 const-does> ( -- n )
  ;

: fconstant ( r "name" -- )
0 1 const-does> ( -- r )
  ;

: simple-field ( n "name" -- )
1 0 const-does> ( addr1 -- addr2 )
  + ;
```

Note that the stack comments after `const-does>` reflect the total stack effect of *name* (including initiation semantics), not the stack effect of the following code.

4 Remarks

The ANS-Forth-style formal proposal may be a bit hard to penetrate, so here are the essentials: `Const-does>` defines a word (the role of `create`) and its behaviour (the role of `does>`). The main other thing it does is to shift *u1* cells and *u2* floats from the definition time of *name* to its execution time. As a consequence, a simple definition like `constant` specifies just how many cells and floats it wants to shift, and needs to do nothing else.

Note that this works for both separate and combined data/FP stacks: On a system with separate stacks `const-does>` shifts *u1* cells and *u2* floats from definition to execution. On system with a combined stack is just shifts as many cells as these cells and floats take.

An optimizing native code compiler could compile a word defined with `const-does>` by compiling the *u1* cells and *u2* floats as literals, and then compiling (and possibly inlining) a call to the code behind the `const-does>`. The compiler would know that these literals are constant, and could optimize accordingly.

An alternative way to create defining words that define optimizable defined words is to use colon definitions and literals. E.g.:

```
: constant ( n "name" -- )
\ name execution: ( -- n )
  >r : r> POSTPONE literal POSTPONE ; ;

: fconstant ( r "name" -- )
\ name execution: ( -- r )
\ environmental dependency: separate FP stack
  : POSTPONE fliteral POSTPONE ; ;

: simple-field ( n "name" -- )
\ name execution: ( addr1 -- addr2 )
  >r : r> POSTPONE literal POSTPONE + POSTPONE ; ;
```

A good compiler could inline the resulting colon definitions and optimize the results.

The downside of this approach is that it is hard to read (even with `]]`...`[[` to eliminate the `POSTPONEs`). And to avoid the environmental dependency for passing floats, we would have to save them in global variables, which would make the code even harder to read.

An alternative syntax would make `const-does>` behave like `does>`, but it would remove the ability to apply `>body` to the execution token of words it is applied to. Advantage: The syntax would be familiar and thus easier to learn. Disadvantages: Optimizing this would require a more complex compiler. The defining word is cluttered with code for saving and restoring the data to be shifted between definition and execution, making the code harder to read. Simple implementations would not enforce the restriction on not applying `>body`, leading to portability bugs (in contrast, the proposed `const-does>` does not expose the placement of the data, so a programmer cannot even access the data without learning implementation details).

One implementation issue for this proposal is that *u1* and *u2* are supplied at run-time. If the numbers were supplied at compile-time (with a syntax like `[ 1 0 ]does-code>`), a native-code compiler using this for optimization could be slightly simpler (and the reference implementation below could be faster). However, a native-code compiler probably can use the same mechanism here that it uses for optimizing `PICK`, and supplying the number at run-time is easier for the user (both syntactically and mentally) and more flexible.

5 Reference implementation

This reference implementation of `const-does>` behaves as it should, but of course does not give you the performance advantages (rather to the contrary).

```
: const-does>-prelude ( u1*x u2*r u1 u2 ''name'' -- )
    \ create name and store u1*x u2*r there
    create 2dup 2,
    over cells allot here >r
    falign dup floats allot here ( u1*x u2*r u1 u2 addr2 )
    swap 0 ?do
        -1 floats + dup f!
    loop
    drop r> ( u1*x u1 addr1 )
    swap 0 ?do
        -1 cells + tuck !
    loop
    drop ;

: const-does>-postlude ( addr -- u1*x u2*r )
    \ fetch u1*x u2*r from addr
    dup 2 cells +
    swap 2@ >r
    0 ?do
        dup @ swap cell+
    loop
    faligned
    r> 0 ?do
        dup f@ float+
    loop
    drop ;

: const-does> ( compilation: colon-sys1 -- colon-sys2 )
    \ run-time: ( u1*x u2*r u1 u2 ''name'' R: nest-sys1 -- )
    \ name initiation: ( -- u1*x u2*r R: nest-sys2 )
    POSTPONE const-does>-prelude
    POSTPONE does>
    POSTPONE const-does>-postlude
; immediate
```

6 Experience

Only the implementation and uses in this paper exist.