Verallgemeinerung von Locals

oder

Variationen über `does>`

M. Anton Ertl
TU Wien

Problem

```
: +field ( u1 u "name" -- u2 )
  create over , +
does> ( addr1 -- addr2 )
  @ + ;

4 4 +field x
: foo x ;
8 ' x >body !

: defword ( a b c "name" -- )
  create , , ,
does>
  >r r@ 2 cells + @ r@ cell+ @ r> @ tu-was ;
```

Lösung 1: Definitionsgeneratoren

```
: +field ( u1 u "name" -- u2 )
  over >r : r> postpone literal postpone + postpone ; + ;

: +field ( u1 u "name" -- u2 )
  over >r : r> ]] literal + ; [[ + ;

: defword ( a b c "name" -- )
  >r >r >r :
  r> ]] literal [[ r> ]] literal [[ r> ]] literal tu-was ; [[ ;
```

\+ Standard-konform

\- Speicherverbrauch

Lösungsvorschlag: `const-data`

```
: +field ( u1 u "name" -- u2 )
  create over , +
  here cell- 1 cells const-data
does> ( addr1 -- addr2 )
  @ + ;

: defword ( a b c "name" -- )
  create , , ,
  here 3 cells - 3 cells const-data
does>
  >r r@ 2 cells + @ r@ cell+ @ r> @ tu-was ;
```

Schon vorhanden in iForth

Lösungsvorschlag: `set-optimizer`

```
: +field1 ( u1 u "name" -- u2 )
  create over , +
does> ( addr1 -- addr2 )
  @ + ;

: +field ( u1 u "name" -- u2 )
  +field1 [: >body @ ]] literal + [[ ;] set-optimizer ;

: +field ( u1 u "name" -- u2 )
  create over , +
  [: @ + ;] set-does> ( addr1 -- addr2 )
  [: >body @ postpone literal postpone + ;] set-optimizer ;

: defword1 ( a b c "name" -- )
  create , , ,
    [: >r r@ 2 cells + @ r@ cell+ @ r> @ tu-was ;] set-does>
    [: >body >r r@ @ r@ cell+ @ r@ 2 cells + @
       ]] literal literal literal tu-was [[ ;] set-optimizer ;
```

Schon vorhanden in Gforth

Lösungsvorschlag: `const-does>`

```
: +field ( u1 u "name" -- u2 )
  over + swap ( u2 u1 )
1 0 const-does> ( u1 )
  + ;

: defword ( a b c "name" -- )
3 0 const-does>
  tu-was ;
```

Lösungsvorschlag: Verallgemeinerte Locals

```
: +field ( u1 u "name" -- u2 )
  create over {: u1 :} +
does> ( addr1 -- addr2 )
  drop u1 + ;

: +field ( u1 u "name" -- u2 )
  create over {: u1 :} +
  [: drop u1 + ;] set-does> ;

: defword ( a b c "name" -- )
  create {: a b c :}
  [: drop a b c tu-was ;] set-does> ;
```

Andere Anwendungen

```
\ numint ( rstart rend xt -- r )
\ xt ( r1 -- r2 )

: myint ( rstart rend rexp -- r )
  {: rexp :} [: ( r1 -- r2 ) rexp fnegate f** ;] numint ;

: 1/y^x {: rx -- xt :}
\ xt ( ry -- r )
  [: ( ry -- r ) rx fnegate f** ;] ;

1e 5e 2e 1/y^x numint
```

Implementierung: Speicherverwaltung

- je nach Lebensdauer
- permanent: `allot creallot`
- Wort-Lebensdauer: Locals/return Stack
- dazwischen: Garbage collection?
- händische Speicherverwaltung
- `allocate free`
- Region

Syntax: Speicherverwaltung

```
: +field ( u1 u "name" -- u2 )
  over [’] creallot <{: u1 :} +
  [: drop u1 + ;] set-does> ;

: myint ( rstart rend rexp -- r )
  {: rexp :} [: ( r1 -- r2 ) rexp fnegate f** ;] numint ;

: 1/y^x ( rx -- handle xt )
  [: allocate throw dup ;] <{: rx :}
  [: ( ry -- r ) rx fnegate f** ;] ;

1e 5e 2e 1/y^x numint f. free
```

Wo sind meine locals?

- Klassisch: Code pointer + environment pointer

- xt hat nur eine Zelle
  Trampolin:
  (temporäres) Wort
  enthält beides
  xt dieses Wortes ist xt des Ganzen

- Speicherverwaltung für Trampolin
  gemeinsam mit Speicher für locals?

Read-only vs. read/write

- read-only locals erlauben Optimierungen
  Eliminieren von Speicherzugriffen
  Replikation

- Erkennen von read-only
  Eventuell eigene Syntax
  `{= a b to c d =}`

Zusammenfassung

- Zugriff auf äußere locals
  hilft u.a. bei `does>`
  aber auch andere Anwendungen

- Explizite Speicherverwaltung für solche locals

- Environment: Trampolines

- read-only locals

- Wann? Fernere Zukunft