Formale Techniken der Software-Entwicklung

Matthias Hölzl, Christian Kroiß

23. Juni 2014

Typen in PVS

- Basistypen: `number`, `nat`, `boolean`, ...
- Aufzählungstypen: `{red, green, blue}`
- Funktionstypen: `[number -> number]`
- Record-Typen: `[# flag: boolean, value: number #]`
- Tupel-Typen: `[boolean, number]`
- Cotupel-Typen (disjunkte Summen): `[boolean + number]`

Typen in PVS

- Algebraische Datentypen (und Codatentypen)

```
list[T: TYPE]: DATATYPE BEGIN
      null: null?
      cons(car: T, cdr: list): cons?
END DATATYPE
```

- Prädikaten-Subtypen:
  - `{x: real | x /= 0}`
  - `{f: [real -> real] | injective?(f)}`
  - `{x: T | P(x)}` kann als `(P)` geschrieben werden
- Strukturelle Subtypen:
  - `[# x,y: real, c: color #]` ist Subtyp von `[# x,y: real #]`
  - Updates respektieren Subtypen
- Abhängige Typen (dependent Types) für Funktionen, Tupel, Records und algebraische Datentypen:

```
[n: nat -> {m: nat | m <= n}]
```

PVS Ausdrücke

- Logik: `TRUE`, `FALSE`, `AND`, `OR`, `XOR`, `NOT`, `IMPLIES`, `FORALL`, `EXISTS`, `=`, ...
- Arithmetik: `+`, `-`, `*`, `/`, `<`, `<=`, `>`, `>=`, `0`, `1`, `2`, ...
- Funktionen:
  - Applikation (`f(x)`)
  - Abstraktion (`LAMBDA (x): A = ...`)
  - Update (`f WITH [(X) := 1]`)
- Hilbert-Operator: `epsilon`
- Typumwandlungen

PVS Ausdrücke

- Records: Konstruktion (`(# size := 0 #)`), Selektion (`size(r)`), Update (`r WITH [size := 1]`)
- Tupel: Konstruktion (`(0, 1)`), Selektion (`proj_1(t)` oder `t`1`), Update (`t WITH [1 := 1]`)
- Konditionale: `IF-THEN-ELSE`, `COND`
- Destrukturierung von Records und Tupeln: (`LET ... = ... IN ...`)
- Pattern-Matching von (Co-)Datentypen `CASES`
- Tabellen

Beispiel: Listen ganzer Zahlen

```
intlist: DATATYPE
 BEGIN
  null: null?
  cons(car: int, cdr: intlist): cons?
 END intlist
```

Beispiel: Listen ganzer Zahlen

PVS generiert zu jedem Datentyp eine umfangreiche Theorie, die die Programmierung und das Beweisen mit solchen Typen erleichtert (in der Datei *datentyp*_adt.pvs.

Typ, Erkenner, Zugriffsfunktionen und Konstruktoren:

```
intlist: TYPE

null?, cons?: [intlist -> boolean]

car: [(cons?) -> int]
cdr: [(cons?) -> intlist]

null: (null?)
cons: [[int, intlist] -> (cons?)]
```

Beispiel: Listen ganzer Zahlen

Abbildung in die ganzen Zahlen:

```
intlist_ord: [intlist -> upto(1)]

intlist_ord_defaxiom: AXIOM
  intlist_ord(null) = 0 AND
   (FORALL (car: int, cdr: intlist):
      intlist_ord(cons(car, cdr)) = 1);

ord(x: intlist): [intlist -> upto(1)] =
    CASES x OF
      null: 0,
      cons(cons1_var, cons2_var): 1
    ENDCASES
```

Beispiel: Listen ganzer Zahlen

Extensionalitätsaxiome („no confusion“)

```
intlist_null_extensionality: AXIOM
  FORALL (null?_var: (null?), null?_var2: (null?)):
    null?_var = null?_var2;

intlist_cons_extensionality: AXIOM
  FORALL (cons?_var: (cons?), cons?_var2: (cons?)):
    car(cons?_var) = car(cons?_var2)
     AND cdr(cons?_var) = cdr(cons?_var2)
       IMPLIES cons?_var = cons?_var2;

intlist_cons_eta: AXIOM
  FORALL (cons?_var: (cons?)):
    cons(car(cons?_var), cdr(cons?_var)) = cons?_var;
```

Beispiel: Listen ganzer Zahlen

Zusammenhang zwischen Konstruktoren und Zugriffsfunktionen:

```
intlist_car_cons: AXIOM
  FORALL (cons1_var: int, cons2_var: intlist):
    car(cons(cons1_var, cons2_var)) = cons1_var;

intlist_cdr_cons: AXIOM
  FORALL (cons1_var: int, cons2_var: intlist):
    cdr(cons(cons1_var, cons2_var)) = cons2_var;
```

Beispiel: Listen ganzer Zahlen

Inklusivität und Induktionsaxiom („no junk“):

```
intlist_inclusive: AXIOM
  FORALL (intlist_var: intlist):
    null?(intlist_var) OR cons?(intlist_var);

intlist_induction: AXIOM
  FORALL (p: [intlist -> boolean]):
    (p(null) AND
      (FORALL (cons1_var: int, cons2_var: intlist):
         p(cons2_var) IMPLIES p(cons(cons1_var, cons2_var))))
     IMPLIES (FORALL (intlist_var: intlist): p(intlist_var));
```

Beispiel: Listen ganzer Zahlen

Nicht-strikte und strikte Subterm-Ordnungen:

```
subterm(x: intlist, y: intlist):  boolean =
    x = y OR
     CASES y
       OF null: FALSE,
          cons(cons1_var, cons2_var): subterm(x, cons2_var)
       ENDCASES;

<<:  (strict_well_founded?[intlist]) =
    LAMBDA (x, y: intlist):
      CASES y
        OF null: FALSE,
           cons(cons1_var, cons2_var): x = cons2_var OR x << cons2_
        ENDCASES;

intlist_well_founded: AXIOM strict_well_founded?[intlist](<<);
```

Beispiel: Listen ganzer Zahlen

Reduktion in die natürlichen Zahlen, wichtig für die Definition des Maßes rekursiver Funktionen:

```
reduce_nat(null?_fun: nat, cons?_fun: [[int, nat] -> nat]):
      [intlist -> nat] =
    LAMBDA (intlist_adtvar: intlist):
      LET red: [intlist -> nat]
            = reduce_nat(null?_fun, cons?_fun) IN
        CASES intlist_adtvar
          OF null: null?_fun,
             cons(cons1_var, cons2_var):
               cons?_fun(cons1_var, red(cons2_var))
          ENDCASES;
```

Beispiel: Listen ganzer Zahlen

Reduktionsfunktion mit allgemeinerer Signatur:

```
REDUCE_nat(null?_fun: [intlist -> nat],
           cons?_fun: [[int, nat, intlist] -> nat]):
      [intlist -> nat] =
    LAMBDA (intlist_adtvar: intlist):
      LET red: [intlist -> nat]
            = REDUCE_nat(null?_fun, cons?_fun) IN
        CASES intlist_adtvar
          OF null: null?_fun(intlist_adtvar),
             cons(cons1_var, cons2_var):
               cons?_fun(cons1_var,
                         red(cons2_var),
                         intlist_adtvar)
          ENDCASES;
```

Beispiel: Listen ganzer Zahlen

Reduktion in die Ordinalzahlen (auch `REDUCE_ordinal`)

```
reduce_ordinal(null?_fun: ordinal,
               cons?_fun: [[int, ordinal] -> ordinal]):
    [intlist -> ordinal] =
  LAMBDA (intlist_adtvar: intlist):
    LET red: [intlist -> ordinal] =
          reduce_ordinal(null?_fun, cons?_fun)
      IN
      CASES intlist_adtvar
        OF null: null?_fun,
           cons(cons1_var, cons2_var):
             cons?_fun(cons1_var, red(cons2_var))
        ENDCASES;
```

Beispiel: Listen ganzer Zahlen

Reduktion in beliebige Bereiche (in eigener Theorie, auch REDUCE)

```
reduce(null?_fun: range, cons?_fun: [[int, range] -> range]):
      [intlist -> range] =
    LAMBDA (intlist_adtvar: intlist):
      LET red: [intlist -> range]
               = reduce(null?_fun, cons?_fun) IN
        CASES intlist_adtvar
          OF null: null?_fun,
             cons(cons1_var, cons2_var):
               cons?_fun(cons1_var, red(cons2_var))
          ENDCASES;
```

Beispiel: Listen ganzer Zahlen

Den Datentyp `intlist` kann man in der Definition von Funktionen und Relationen verwenden:

```
length(is: intlist): RECURSIVE nat =
  CASES is OF
    null: 0,
    cons(j, js): 1 + length(js)
    ENDCASES
  MEASURE reduce_nat(0, lambda(k: int, l: nat): 1 + l)

length_is_reduce_plus1: LEMMA
  forall (is: intlist):
    length(is)
      = reduce(0, lambda(i: int, n: nat): 1 + n)(is)
```

Beispiel: Listen ganzer Zahlen

```
plus1(i: int, n: nat): nat = 1 + n
length_is_reduce_plus1: LEMMA
  forall (is: intlist):
    length(is) = reduce(0, plus1)(is)
```

Ein Beweis für das Lemma ist z.B. folgender:

```
(""
 (induct "is")
 (("1" (expand "length") (expand "reduce") (propax))
  ("2"
   (skolem!)
   (flatten)
   (expand "length" 1)
   (expand "reduce" 1)
   (propax))))
```

Beispiel: Listen ganzer Zahlen

Auch Funktionen höherer Ordnung lassen sich leicht spezifizieren/implementieren und durch Beweise testen:

```
map(f: [int -> int], is: intlist): RECURSIVE intlist =
  CASES is OF
    null: null,
    cons(j, js): cons(f(j), map(f, js))
  ENDCASES
  MEASURE length(is)

map_is_reduce_cons: LEMMA
  forall (is: intlist): forall (f: [int -> int]):
    map(f, is) = reduce(null,
                        lambda(j: int, js: intlist):
                            cons(f(j), js))(is)
```

Beispiel: Listen ganzer Zahlen

```
map_is_reduce_cons: LEMMA
  forall (is: intlist): forall (f: [int -> int]):
    map(f, is) = reduce(null,
                        lambda(j: int, js: intlist):
                            cons(f(j), js))(is)
```

Auch hier ist es leich mit PVS einen Beweis zu finden:

```
(""
 (induct "is")
 (("1" (skosimp*) (expand "map")
       (expand "reduce") (propax))
  ("2" (skosimp*) (expand "map" 1)
       (expand "reduce" 1) (rewrite -1))))
```

(Mit `induct-and-simplify` findet PVS den Beweis auch vollautomatisch.)

Beispiel: Listen ganzer Zahlen

Ein Beispiel für die Anwendung der `map`-Funktion:

```
map_example: LEMMA
  map(lambda(n: nat): n * n,
      cons(1, cons(2, cons(3, cons(4, null)))))
  =
  cons(1, cons(4, cons(9, cons(16, null))))
```

Beispiel: Listen beliebiger Elemente

Datentypen wie Listen sollen in Spezifikationen, genau wie in Programmiersprachen, für verschiedene Elementtypen einsetzbar sein. PVS bietet dafür parametrische Theorien (und Theorieinterpretationen) an:

```
plist[T: TYPE]: DATATYPE
 BEGIN
  null: null?
  cons(car: T, cdr: plist): cons?
 END plist
```

Beispiel: Listen beliebiger Elemente

Auch hier generiert PVS die für `intlist` besprochene Theorie. Natürlich ist die generierte Theorie ebenfalls parametrisch:

```
plist_adt[T: TYPE]: THEORY
 BEGIN
  plist: TYPE
  null?, cons?: [plist -> boolean]

  car: [(cons?) -> T]
  cdr: [(cons?) -> plist]

  null: (null?)
  cons: [[T, plist] -> (cons?)]

  ...
 end
```

Beispiel: Listen beliebiger Elemente

In diesem Fall werden auch Definitionen für `every` und `some` generiert:

```
every(p: PRED[T], a: plist):  boolean =
    CASES a
      OF null: TRUE,
         cons(cons1_var, cons2_var):
             p(cons1_var) AND every(p, cons2_var)
      ENDCASES;

some(p: PRED[T])(a: plist):  boolean =
    CASES a
      OF null: FALSE,
         cons(cons1_var, cons2_var):
            p(cons1_var) OR some(p)(cons2_var)
      ENDCASES;
```

Beispiel: Geordnete Binärbäume

Noch ein Beispiel für DATATYPE: Eine parametrische Datenstruktur für Binärbäume

```
binary_tree[T : TYPE] : DATATYPE
BEGIN
  leaf : leaf?
  node(val : T,
       left : binary_tree,
       right : binary_tree) : node?
END binary_tree
```

Beispiel: Geordnete Binärbäume

Auch diese Definition generiert eine Typdeklaration, Erkennungsfunktionen (Recognizers), Konstruktoren und Zugriffsfunktionen:

```
binary_tree: TYPE

leaf?: [binary_tree -> boolean]
leaf: (leaf?)

node?: [binary_tree -> boolean]
node: [T, binary_tree, binary_tree -> (node?)]

val: [(node?) -> T]
left: [(node?) -> binary_tree]
right: [(node?) -> binary_tree]
```

Beispiel: Geordnete Binärbäume

Mit DATATYPE deklarierte Typen sind initiale Algebren: für sie gelten Extensionalität („no confusion“) und ein Induktionsschema („no junk“):

```
binary_tree_node_extensionality: AXIOM
  FORALL (node?_var: (node?), node?_var2: (node?)):
    val(node?_var) = val(node?_var2) AND
     left(node?_var) = left(node?_var2) AND
      right(node?_var) = right(node?_var2)
     IMPLIES node?_var = node?_var2;

binary_tree_induction: AXIOM
  FORALL (p: [binary_tree -> boolean]):
    (p(leaf) AND
      (FORALL (node1_var: T, node2_var: binary_tree,
               node3_var: binary_tree):
          p(node2_var) AND p(node3_var) IMPLIES
           p(node(node1_var, node2_var, node3_var))))
     IMPLIES (FORALL (binary_tree_var: binary_tree):
                p(binary_tree_var));
```

Beispiel: Geordnete Binärbäume

Die Funktion `reduce_nat` wird in den folgenden Definitionen zur Definition des Maßes verwendet und wird von PVS aus der `DATATYPE` Deklaration generiert:

```
reduce_nat(leaf?_fun: nat, node?_fun: [[T, nat, nat] -> nat]):
    [binary_tree -> nat] =
  LAMBDA (binary_tree_adtvar: binary_tree):
    CASES binary_tree_adtvar OF
      leaf: leaf?_fun,
      node(node1_var, node2_var, node3_var):
          node?_fun(node1_var,
                    reduce_nat(leaf?_fun, node?_fun)(node2_var),
                    reduce_nat(leaf?_fun, node?_fun)(node3_var))
    ENDCASES;
```

Beispiel: Geordnete Binärbäume

Geordnete Binärbäume können als Theorie definiert werden, die sowohl im Wertetyp als auch in der Ordnungsrelation parametrisch ist:

```
obt [T : TYPE,  <= : (total_order?[T])] : THEORY
BEGIN
IMPORTING binary_tree[T]

A, B, C: VAR binary_tree
x, y, z: VAR T
pp: VAR pred[T]
i, j, k: VAR nat

...
END obt
```

Beispiel: Geordnete Binärbäume

Um ein Maß für die rekursiven Funktionen angeben zu können definieren wir die Funktion `size`. Die Funktion `ordered?` überprüft, ob alle Werte im linken Teilbaum kleiner als der Wert des Knotens, und alle Werte im rechten Teilbaum größer als der Wert des Knotens sind.

```
size(A) : nat =
  reduce_nat(0, (LAMBDA (x, i, j): i + j + 1))(A)

ordered?(A) : RECURSIVE bool =
  (IF node?(A) THEN (every((LAMBDA y: y<=val(A)), left(A)) AND
                     every((LAMBDA y: val(A)<=y), right(A)) AND
                     ordered?(left(A)) AND
                     ordered?(right(A)))
        ELSE TRUE ENDIF)
  MEASURE size
```

Beispiel: Geordnete Binärbäume

Einfügen erfolgt wie üblich durch Vergleich mit dem Wert an der Wurzel und Rekursion in den linken oder rechten Unterbaum:

```
insert(x, A): RECURSIVE binary_tree[T] =
  (CASES A OF
    leaf: node(x, leaf, leaf),
    node(y, B, C): (IF x<=y THEN node(y, insert(x, B), C)
                            ELSE node(y, B, insert(x, C))
                    ENDIF)
   ENDCASES)
  MEASURE size(A)
```

Hier sieht man die Verwendung von `CASES` mit Pattern-Matching über den Konstruktoren von `binary_tree`.

Beispiel: Geordnete Binärbäume

Das folgende Lemma beschreibt eine einfache Eigenschaft des Einfügeschritts: wenn `x` das Prädikat `pp` erfüllt und jedes Element aus `A` das Prädikat `pp` erfüllt, dann erfüllt auch jedes Element aus dem durch Einfügen von `x` in `A` entstandenen Baum `pp`:

```
ordered?_insert_step: LEMMA
   pp(x) AND every(pp, A) IMPLIES every(pp, insert(x, A))
```

Beispiel: Geordnete Binärbäume

Das folgende Theorem zeigt die Korrektheit des `insert`-Algorithmus:

```
ordered?_insert: THEOREM
   ordered?(A) IMPLIES ordered?(insert(x, A))
```

Beweis:

```
(""
 (induct-and-simplify "A" :rewrites "ordered?_insert_step")
 ;; Rewriting mit I-A-S (oder GRIND, etc.) ist stärkter als
 ;; Induktion und Rewriting separat:
 ;;   (induct-and-simplify "A")
 ;;   (rewrite "ordered?_insert_step")
 ;; ist nicht ausreichend um die Aussage zu beweisen.
 (typepred "<=")
 (grind :if-match all))
```

Beispiel: Geordnete Binärbäume

Binärsuche lässt sich ebenfalls leicht implementieren:

```
search(x, A): RECURSIVE bool =
  (CASES A OF
    leaf: FALSE,
    node(y, B, C): (IF x = y THEN TRUE
                    ELSIF x<=y THEN search(x, B)
                      ELSE search(x, C)
                    ENDIF)
   ENDCASES)
  MEASURE size(A)
```

Beispiel: Geordnete Binärbäume

Das folgende Theorem besagt, dass `search` und `insert` auf die gewünschte Weise interagieren:

```
search_insert: THEOREM
  search(y, insert(x, A)) = (x = y OR search(y, A))
```

(Beweis mit `induct-and-simplify`.)

Beispiel: Aussagenlogik

Wir hatten die Semantik aussagenlogischer Formeln folgendermaßen definiert:

$$
\begin{aligned}
[\![\top]\!]\eta &= \text{wahr} \\
[\![\bot]\!]\eta &= \text{falsch} \\
[\![\alpha]\!]\eta &= \eta(\alpha) \quad \text{für } \alpha \in \mathcal{A} \\
[\![\neg\phi]\!]\eta &= !([\![\phi]\!]\eta) \\
[\![\phi \wedge \psi]\!]\eta &= [\![\phi]\!]\eta \,\&\, [\![\psi]\!]\eta \\
[\![\phi \vee \psi]\!]\eta &= [\![\phi]\!]\eta \mid [\![\psi]\!]\eta \\
[\![\phi \implies \psi]\!]\eta &= !([\![\phi]\!]\eta) \mid [\![\psi]\!]\eta \\
[\![\phi \iff \psi]\!]\eta &= ([\![\phi]\!]\eta \,\&\, [\![\psi]\!]\eta) \mid (!([\![\phi]\!]\eta) \,\&\, !([\![\psi]\!]\eta))
\end{aligned}
$$

Beispiel: Aussagenlogik

Das können wir auch durch eine PVS-Spezifikation ausdrücken:

```
pl_formula: DATATYPE
 BEGIN
  lvar(name: string): lvar?
  lnot(arg: pl_formula): lnot?
  land(lhs, rhs: pl_formula): land?
  lor(lhs, rhs: pl_formula): lor?
  limp(lhs, rhs: pl_formula): limp?
  liff(lhs, rhs: pl_formula): liff?
 END pl_formula
```

Beispiel: Aussagenlogik

```
valuation: TYPE = [(lvar?) -> bool]
plus1(m, n: nat): nat = m + n + 1

eval(p: pl_formula, v: valuation): RECURSIVE bool =
  CASES p
    OF lvar(name): v(p),
       lnot(arg): NOT (eval(arg, v)),
       land(lhs, rhs): eval(lhs, v) AND eval(rhs, v),
       lor(lhs, rhs): eval(lhs, v) OR eval(rhs, v),
       limp(lhs, rhs): NOT (eval(lhs, v)) OR eval(rhs, v),
       liff(lhs, rhs): eval(lhs, v) = eval(rhs, v)
    ENDCASES
   MEASURE LAMBDA (p: pl_formula, v: valuation):
             reduce_nat(LAMBDA (name: string): 1,
                        LAMBDA (n: nat): n + 1,
                        plus1, plus1, plus1, plus1)
                       (p)
```

Beispiel: Aussagenlogik

Wir hatten verschiedene Belegungen definiert...

$$\eta_T : \mathcal{A} \to \mathbb{B} \qquad \eta_F : \mathcal{A} \to \mathbb{B}$$
$$\alpha \mapsto \text{wahr} \qquad \alpha \mapsto \text{falsch}$$

$$\eta_T(A) = \text{wahr} \qquad \eta_F(A) = \text{falsch}$$
$$\eta_T(B) = \text{wahr} \qquad \eta_F(B) = \text{falsch}$$
$$\ldots \qquad \ldots$$

$$\eta_A : \mathcal{A} \to \mathbb{B}$$

$$\alpha \mapsto \begin{cases} \text{wahr} & \text{falls } \alpha = A \\ \text{falsch} & \text{sonst} \end{cases}$$

$$\eta_A(A) = \text{wahr}$$
$$\eta_A(B) = \text{falsch}$$
$$\eta_A(C) = \text{falsch}$$
$$\ldots$$

Beispiel: Aussagenlogik

. . . und gezeigt, dass sie bei manchen Formeln zu verschiedenen Wahrheitswerten führen:

$$\begin{aligned}
[\![A \wedge B]\!]\eta_T &= [\![A]\!]\eta_T \,\&\, [\![B]\!]\eta_T \\
&= \eta_T(A) \,\&\, \eta_T(B) \\
&= \text{wahr} \,\&\, \text{wahr} \\
&= \text{wahr}
\end{aligned}$$

$$\begin{aligned}
[\![A \wedge B]\!]\eta_A &= [\![A]\!]\eta_A \,\&\, [\![B]\!]\eta_A \\
&= \eta_A(A) \,\&\, \eta_A(B) \\
&= \text{wahr} \,\&\, \text{falsch} \\
&= \text{falsch}
\end{aligned}$$

Beispiel: Aussagenlogik

In PVS: Die Belegungen...

```
v_false: valuation =
    lambda(lvar: (lvar?)): false

v_true: valuation =
    lambda(lvar: (lvar?)): true

v_a: valuation =
    v_false WITH [(lvar("A")) := true]
```

Beispiel: Aussagenlogik

. . . und die Auswertung von Formeln:

```
p_1: pl_formula = land(lvar("A"), lvar("B"))

p_1_may_be_false: LEMMA eval(p_1, v_false) = false

p_1_may_be_true: LEMMA eval(p_1, v_true) = true
```

Beispiel: Aussagenlogik

Wir können in PVS auch Aussagen über die Allgemeingültigkeit oder Unerfüllbarkeit von Formeln machen:

```
p_2: pl_formula = land(lvar("A"), lnot(lvar("A")))

p_2_is_always_false: LEMMA
    forall(v: valuation): eval(p_2, v) = false
```

Beweis:

```
(""
 (SKOLEM!) (EXPAND "p_2") (EXPAND "eval")
 (FLATTEN) (EXPAND "eval") (EXPAND "eval")
 (PROPAX))
```

(oder einfach `(grind)`)

Beispiel: Virtuelle Maschine

Wir spezifizieren eine einfache stack-basierte virtuelle Maschine mit PVS, die z.B. Programme der folgenden Art ausführen kann:

```
(comp-show '(lambda (x) (+ x 1)))
   0: ARGS    0
   1: FN
            0: ARGS    1
            1: LVAR    0       0       ;       X
            2: 1
            3: +
            4: RETURN
   2: RETURN
```

Beispiel: Virtuelle Maschine

```
(comp-show '((lambda (x) (+ x 1)) 7))
   0: ARGS    0
   1: CONST   7
   2: FN
          0: ARGS    1
          1: LVAR    0       0       ;       X
          2: 1
          3: +
          4: RETURN
   3: CALLJ   1
```

Beispiel: Virtuelle Maschine

```
(comp-show '(define (foo x y) (+ x y)))
   0: ARGS    0
   1: FN
          0: ARGS    2
          1: LVAR    0       0       ;       X
          2: LVAR    0       1       ;       Y
          3: +
          4: RETURN
   2: GSET    FOO
   3: CONST   FOO
   4: SET-NAME!
   5: RETURN
```

Beispiel: Virtuelle Maschine

```
(comp-show '(define (foo x) (lambda (y) (+ x y))))
   0: ARGS    0
   1: FN
          0: ARGS    1
          1: FN
                 0: ARGS    1
                 1: LVAR    1       0       ;       X
                 2: LVAR    0       0       ;       Y
                 3: +
                 4: RETURN
          2: RETURN
   2: GSET    FOO
   3: CONST   FOO
   4: SET-NAME!
   5: RETURN
```