

## Funktionsanwendung:

$$\eta x_1 = (\mathbf{fn} x \Rightarrow e, \eta_1)$$

$$\eta x_2 = (v_2, \eta_2)$$

$$(e, \eta_1 \oplus \{x \mapsto (v_2, \eta_2)\}) \Longrightarrow (v_3, \eta_3)$$

---

$$(x_1 x_2, \eta) \Longrightarrow (v_3, \eta_3)$$

## Lokal rekursive Funktionsanwendung:

$$\eta x_1 = (\text{fix}(f, \mathbf{fn} x \Rightarrow e), \eta_1)$$

$$\eta x_2 = (v_2, \eta_2)$$

$$(e, \eta_1 \oplus \{f \mapsto (\text{fix}(f, \mathbf{fn} x \Rightarrow e), \eta_1), x \mapsto (v_2, \eta_2)\}) \Longrightarrow (v_3, \eta_3)$$

---

$$(x_1 \ x_2, \eta) \Longrightarrow (v_3, \eta_3)$$

## Fall-Unterscheidung 1:

$$\eta x = (b, \eta_1)$$

$$(e_i, \eta) \Longrightarrow (v_i, \eta_i)$$

---

$$(\mathbf{case } x \mathbf{ of } p_1 : e_1 \mid \dots \mid p_k : e_k \mathbf{ end}, \eta) \Longrightarrow (v_i, \eta_i)$$

sofern  $p_i \equiv b$  das erste auf  $b$  passende Muster ist :-)

## Fall-Unterscheidung 2:

$$\eta x = (c z, \eta_1)$$

$$(e_i, \eta \oplus \{x_i \mapsto (\eta z)\}) \Longrightarrow (v_i, \eta_i)$$

---

$$(\mathbf{case } x \mathbf{ of } p_1 : e_1 \mid \dots \mid p_k : e_k \mathbf{ end}, \eta) \Longrightarrow (v_i, \eta_i)$$

sofern  $p_i \equiv c x_i$  das erste auf  $c z$  passende Muster ist :-)

## Fall-Unterscheidung 3:

$$\eta x = ((z_1, \dots, z_m), \eta_1)$$

$$(e_i, \eta \oplus \{y_j \mapsto (\eta z_j) \mid j = 1, \dots, m\}) \implies (v_i, \eta_i)$$

---

$$(\mathbf{case } x \mathbf{ of } p_1 : e_1 \mid \dots \mid p_k : e_k \mathbf{ end}, \eta) \implies (v_i, \eta_i)$$

für das erste passende Muster  $p_i \equiv (y_1, \dots, y_m) \quad :-)$

## Fall-Unterscheidung 4:

$$(e_i, \eta \oplus \{x_i \mapsto (\eta x)\}) \Longrightarrow (v_i, \eta_i)$$

---

$$(\mathbf{case } x \mathbf{ of } p_1 : e_1 \mid \dots \mid p_k : e_k \mathbf{ end}, \eta) \Longrightarrow (v_i, \eta_i)$$

sofern  $p_i \equiv x_i$  und alle Muster davor **fehl** schlugen :-)

## Lokale Definitionen:

$$(e_1, \eta) \Longrightarrow (v_1, \eta_1)$$

$$(e_2, \eta \oplus \{x_1 \mapsto (v_1, \eta_1)\}) \Longrightarrow (v_2, \eta_2)$$

...

$$(e_k, \eta \oplus \{x_1 \mapsto (v_1, \eta_1), \dots, x_{k-1} \mapsto (v_{k-1}, \eta_{k-1})\}) \Longrightarrow (v_k, \eta_k)$$

$$(e_0, \eta \oplus \{x_1 \mapsto (v_1, \eta_1), \dots, x_k \mapsto (v_k, \eta_k)\}) \Longrightarrow (v_0, \eta_0)$$

---

$$(\mathbf{let } x_1 = e_1 \dots x_k = e_k \mathbf{ in } e_0 \mathbf{ end}, \eta) \Longrightarrow (v_0, \eta_0)$$

Variablen:

$$\eta(x) = (v_1, \eta_1)$$

---

$$(x, \eta) \implies (v_1, \eta_1)$$



## Korrektheit der Analyse:

Man zeigt für jedes  $(e, \eta)$ , das in einer Ableitung für das Programm vorkommt:

- Falls  $\eta(x) = (v, \eta_1)$ , dann ist  $v \in \llbracket x \rrbracket^\sharp$ .
- Falls  $(e, \eta) \Longrightarrow (v, \eta_1)$ , dann ist  $v \in \llbracket e \rrbracket^\sharp$ .

## Fazit:

$\llbracket e \rrbracket^\sharp$  liefert eine **Obermenge** der Werte, zu denen sich  $e$  möglicherweise ausrechnet :-)

## 4.4 Anwendung: Inlining

### Probleme:

- globale Variablen. Das Programm:

```
let  x = 1
    f = let  x = 2
        in  fn y => y + x
    end
in   f x
end
```

... berechnet offenbar etwas anderes als:

```
let  x = 1
    f = let  x = 2
          in  fn y => y + x
        end
in    let  y = x
      in  y + x
      end
end
```

- rekursive Funktionen. In der Definition:

$$x = \text{fix} (\text{foo}, \text{fn } y \Rightarrow \text{foo } y)$$

sollten wir `foo` besser nicht substituieren :-)

## Idee 1:

- Wir machen erstmal die Namen im Programm **eindeutig**.
- Dann substituieren wir nur Funktionen, die **statisch** im Scope der **selben** globalen Variablen stehn, wie die Anwendung **:-)**
- Wir berechnen für jeden Ausdruck alle Funktions-Definitionen mit dieser Eigenschaft **:-)**

Sei  $D[e]$  die Menge der Definitionen, die in  $e$  statisch ankommen.

- Für  $e \equiv \mathbf{let} \ x_1 = e_1 \ \dots \ x_k = e_k \ \mathbf{in} \ e_0 \ \mathbf{end}$  haben wir:

$$D[e_1] = D$$

...

$$D[e_k] = D \cup \{x_1, \dots, x_{k-1}\}$$

$$D[e_0] = D \cup \{x_1, \dots, x_k\}$$

- In den anderen Fällen propagiert sich  $D$  unverändert zu den Teilausdrücken :-)

Für  $e \equiv \mathbf{fn} \ x \Rightarrow e_1$  haben wir etwa:

$$D[e_1] = D$$

... im Beispiel:

```
let  x = 1
    f = let  x1 = 2
          in  fn y ⇒ y + x1
        end
in    f x
end
```

... steht (nach Umbenennung :-))  $f$  für  $f x$  statisch zur Verfügung. Bei Substitution erhalten wir:

```

let   $x = 1$ 
       $f = \text{let } x_1 = 2$ 
          in  fn  $y \Rightarrow y + x_1$ 
              end
in
    let   $y = x$ 
        in  let   $x_1 = 2$ 
            in   $y + x_1$ 
                end
        end
    end
end

```

Ersetzen der Variablen-Variablen-Umbenennungen ergibt schließlich:

```
let  x = 1
    f = let  x1 = 2
          in  fn y ⇒ y + x1
        end
    in
      let  x1 = 2
          in  x + x1
          end
    end
end
```



## Idee 2:

- Wir benutzen unsere Wert-Analyse.
- Wir **ignorieren** globale Variablen :-)
- Wir substituieren nur Funktionen **ohne** freie Variablen :-))

Beispiel: Die **map**-Funktion

```

let  $f = \text{fn } x \Rightarrow x \cdot x$ 
       $\text{map} = \text{fix}(\text{map}, \text{fn } g \Rightarrow \text{fn } x \Rightarrow \text{case } x$ 
          of  $[] : []$ 
          |  $:: z : \text{case } z \text{ of } (x_1, x_2) \text{ in}$ 
              let  $y_1 = g \ x_1$ 
                   $m = \text{map } g$ 
                   $y_2 = m \ x_2$ 
                   $z_1 = (y_1, y_2)$ 
              in  $:: z_1$ 
              end
          end)
       $h = \text{map } f$ 
in  $h \ \text{list}$ 
end

```

- Der **formale** Parameter  $g$  von **map** ist stets  $f$  :-)
- Wir können die Anwendung von  $f$  in der Definition von **map** ersetzen:

```

map = fix (map, fn g => fn x => case x
  of [] : []
  | :: z : case z of (x1, x2) in
    let y1 = let x = x1
              in x · x
            end
            m = map g
            y2 = m x2
            z1 = (y1, y2)
          in :: z1
        end
    end)
h = map f

```

- Noch mehr könnten wir sparen, wenn wir die **spezialisierte** Funktion  $h = \text{map } f$  direkt definieren könnten :-)
- Dazu müssen wir **überall** in der Definition von **map** das Muster `map g` durch  $h$  ersetzen ...

$\implies$  **fold-Transformation** :-)

- Alle weiteren Vorkommen von  $g$  müssen durch (die Definition von)  $f$  ersetzt werden ...  
`//` kommt hier nicht vor :-)

```

map = fix (
  map, fn g => fn x => case x
    of [] : []
      | :: z : case z of (x1, x2) in
          let y1 = let x = x1
                    in x · x
                  end
              m = map g
              y2 = m x2
              z1 = (y1, y2)
            in :: z1
          end
        end)
h = map f

```

```

h = fix (h, fn x ⇒ case x
  of [] : []
  |   :: z : case z of (x1, x2) in
    let y1 = let x = x1
              in x · x
              end
    m = h
    y2 = m x2
    z1 = (y1, y2)
  in :: z1
end
end)

```

Beseitigung von Variablen-Variablen-Umspeicherungen liefert:

```
h = fix(h, fn x ⇒ case x
  of [] : []
  | :: z : case z of (x1, x2) in
    let y1 = x1 · x1
        y2 = h x2
        z1 = (y1, y2)
    in :: z1
  end
end)
```