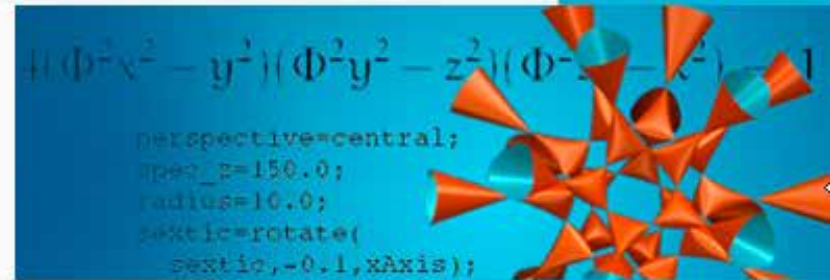


Deklarative Programmierung

Sommersemester 2018

Prof. Christoph Bockisch
(Programmiersprachen und –werkzeuge)
Steffen Dick, Alexander Bille, Johannes Frankenau,
Patrick Frömel, Niclas Schmidt, Jonas Stettin,
Robert Tran, Julian Velten



[Skript 12]

Sprachniveau ISL+

- Intermediate Student Language Plus (ISL+)
- Erweitert BSL um einige Konstrukte
 - Lokale Definitionen
 - Funktionen als Werte (mit Umgebung) betrachten (Closures)

Bedeutung von ISL+

- Definition der Bedeutung wie bei BSL
 - Kernsprache
 - Syntax als EBNF Grammatik
 - Semantik entsprechend der Grammatik-Regeln
 - Auswertungsregeln
 - Reduktionsrelation
- Keine erneute Definition der Bedeutung für Sprachkonzepte, die durch neue Konzepte nicht beeinflusst werden
 - Strukturdefinitionen (define-struct)
 - Konditionale Ausdrücke (cond)
 - Logische Operatoren (and)
- Vereinfachung durch uniforme Behandlung von Funktionen und Werten
 - Funktionsdefinition ist syntaktischer Zucker für Definition einer Konstanten mit Lambda-Ausdruck

Syntax der Kernsprache

$\langle \text{definition} \rangle ::= (' \text{define} \langle \text{name} \rangle \langle \text{e} \rangle ')$

$\langle \text{e} \rangle ::= (' \langle \text{e} \rangle \langle \text{e} \rangle^+ ')$

| $(' \text{local} '[' \langle \text{definition} \rangle^+ ']' \langle \text{e} \rangle ')$

| $\langle \text{name} \rangle$

| $\langle \text{v} \rangle$

$\langle \text{v} \rangle ::= (' \text{lambda} (' \langle \text{name} \rangle^+ ') \langle \text{e} \rangle ')$

| $\langle \text{number} \rangle$

| $\langle \text{boolean} \rangle$

| $\langle \text{string} \rangle$

| $\langle \text{image} \rangle$

| $' + '$

| $' * '$

| ...

Namen aller
primitiven
Funktionen

Umgebung

- Durch Wegfall von Funktionsdefinitionen (und Auslassung von Strukturdefinitionen) vereinfacht sich die Umgebung
- Nur noch Sequenz von Konstantendefinitionen

$\langle \text{env} \rangle ::= \langle \text{env-element} \rangle^*$

$\langle \text{env-element} \rangle ::= \text{'(' 'define' } \langle \text{name} \rangle \langle \text{v} \rangle \text{'('}$

Bedeutung von Programmen

- **(PROG):** Ein Programm wird von links nach rechts ausgeführt und startet mit der leeren Umgebung. Ist das nächste Programmelement ...
 - ... eine Funktionsdefinition, so wird diese Definition in die Umgebung aufgenommen und diese Definition mit dem nächsten Element fortgesetzt
Identisch zu bisheriger Definition bis auf den Wegfall des Falls "Struktur- oder Funktions-Definition"
 - ... ein Ausdruck, so wird dieser gemäß der nachfolgenden Regeln in der aktuellen Umgebung zu einem Wert ausgewertet.
 - ... eine Konstantendefinition (`define x e`), so wird in der aktuellen Umgebung zunächst `e` zu einem Wert `v` ausgewertet und dann (`define x v`) zur aktuellen Umgebung hinzugefügt.

Bedeutung von Programmen

- **(PROG):** Ein Programm wird von links nach rechts ausgeführt und startet mit der leeren Umgebung. Ist das nächste Programmelement ...
 - ... eine Funktions- oder Strukturdefinition, so wird diese Definition in die Umgebung aufgenommen und die Ausführung mit dem nächsten Programmelement fortgesetzt.
 - ... eine lokale Definition (LOCAL) ist, so wird durch die später definierte Auswertungsregel (LOCAL) beeinflusst, was das "nächste Programmelement" ist.
 - ... eine Konstantendefinition (`define x e`), so wird in der aktuellen Umgebung zunächst `e` zu einem Wert `v` ausgewertet und dann (`define x v`) zur aktuellen Umgebung hinzugefügt.

Auswertungspositionen und die Kongruenzregel

- Im Gegensatz zu BSL tritt auch an der ersten Position eines Funktionsaufrufs ein Ausdruck auf

$\langle E \rangle ::= '[]'$
 $| '(\langle v \rangle^* \langle E \rangle \langle e \rangle^* ')'$

Hier stand bei BSL der Funktionsname. Jetzt kann an erster Position ein Closure-Wert stehen.

- Die Kongruenzregel gilt unverändert
 - (KONG): Falls $e_1 \rightarrow e_2$, dann $E[e_1] \rightarrow E[e_2]$.

Bedeutung von Funktionsaufrufen

- Es gibt keine Funktionsdefinitionen mehr
 - Stattdessen: Konstanten mit Lambda als Wert
 - Konstantenname, der für die Funktion steht wird ausgewertet zu Lambda-Ausdruck
 - Anstelle eines Funktionsnamens: Lambda-Ausdruck an erster Position
- Daher:
 - Wegfall der Regel (FUN) für Funktionsaufrufe
 - Neu: Regel (APP) für Anwendung eines Lambda-Ausdrucks

Bedeutung von Funktionsaufrufen

- **(APP):** $((\text{lambda } (name_1 \dots name_n) e) v_1 \dots v_n) \rightarrow e[name_1 := v_1 \dots name_n := v_n]$
- Ersetzung der formalen Parameter muss Scoping-Regeln beachten
 - Es können nicht einfach alle Vorkommen des Bezeichners ersetzt werden
 - Beispiel:
 - $((\text{lambda } (x) (+ x 1)) (* x 2))[x := 7] = ((\text{lambda } (x) (+ x 1)) (* 7 2))$

Dies sind unterschiedliche Variablen mit demselben Namen, aber in unterschiedlichen Scopes.

In diesem Fall darf nur das x im äußeren Scope ersetzt werden.

Bedeutung von Funktionsaufrufen

- Im Fall, dass an erster Position eines Ausdrucks der Name einer primitiven Funktion steht, gilt (PRIM) analog der bisherigen Definition
 - **(PRIM):** Falls v eine primitive Funktion f ist und $f(v_1, \dots, v_n) = v'$, dann $(v \ v_1 \ \dots \ v_n) \rightarrow v'$.

Hier steht ein Wert der einer primitiven Funktion entspricht anstelle des Funktionsnamens wie bisher.

Auch primitive Funktionen können das Resultat von Berechnungen sein, bspw.:
 $((\text{if } \langle \text{cond} \rangle + *) \ 3 \ 4)$

Bedeutung von lokalen Definitionen

- Unterschied zu globalen Definitionen
 - Eingeschränkter Gültigkeitsbereich
 - Zugriff auf lokalen Kontext
- Auswertung ist dynamisch!
 - Während der Auswertung können sie ähnlich wie globale Definitionen behandelt werden
- Unterschiede:
 - Gültigkeitsbereich spielt eine Rolle bei der Suche nach einer Definition
 - Lokaler Kontext ist während der Auswertung zugreifbar

Bedeutung von lokalen Definitionen

- Für die Auswertung:
 - Ersetzen von lokaler durch globale Definition
 - Zugriff auf lokalen Kontext wurde bereits vorher durch Regeln wie (APP) oder (CONST) ersetzt ("Substitution")
- (LOCAL):

$$E[(\text{local } [(\text{define } name_1 e_1) \dots (\text{define } name_n e_n)] e)]$$

$$\rightarrow (\text{define } name_1' e_1') \dots (\text{define } name_n' e_n') E[e]$$

wobei $name_1', \dots, name_n'$ "frische" Namen sind die sonst nirgendwo im Programm vorkommen
 und e', e_1', \dots, e_n' Kopien von e, e_1, \dots, e_n sind, in denen alle Vorkommen von $name_1, \dots, name_n$ durch $name_1', \dots, name_n'$ ersetzt werden.

Bedeutung von lokalen Definitionen

- Für die Auswertung:
 - Ersetzen von lokaler durch globale Definition
 - Zugriff auf lokalen Kontext wurde bereits vorher durch Regel (APP) ersetzt ("Substitution")

- (LOCAL):

$$E[(\text{local } [(\text{define } name_1 e_1) \dots (\text{define } name_n e_n)] e)] \rightarrow (\text{define } name_1' e_1') \dots (\text{define } name_n' e_n') E[e']$$

wobei $name_1', \dots, name_n'$ "frische" Namen sind die sonst nirgends vorkommen und e_1', \dots, e_n' die Auswertung von e_1, \dots, e_n sind.

Die lokalen Definitionen werden wie globale behandelt und werden zum "nächsten Programmelement" für die Auswertung gemäß (PROG). Nach Abarbeitung der Definitionen wird die Auswertung $E[e']$ fortgesetzt.

Bedeutung von lokalen Definitionen

- Für die Auswertung:
 - Ersetzen von lokaler durch globale Definition
 - Zugriff auf lokalen Kontext wurde bereits vorher durch Regel (APP) ersetzt ("Substitution")

Namen werden so ersetzt, dass sie nur innerhalb des Scope des local-Ausdrucks verwendet werden.

- (LOCAL $E[(\text{local } name_1 e_1) \dots (\text{local } name_n e_n)] e]$
 $\rightarrow (\text{define } name_1' e_1') \dots (\text{define } name_n' e_n') E[e]$

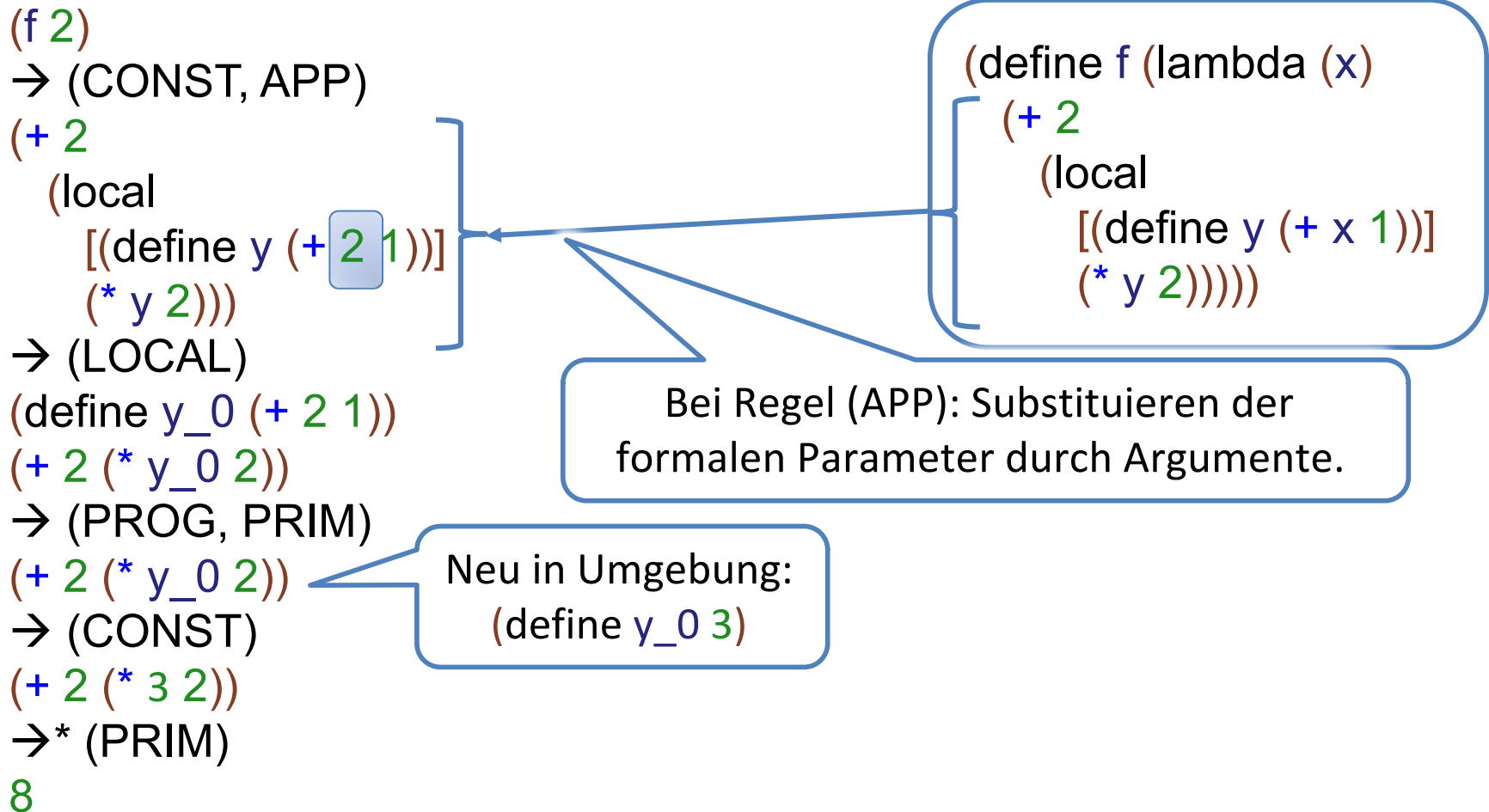
wobei $name_1', \dots, name_n'$ "frische" Namen sind die sonst nirgendwo im Programm vorkommen und e', e_1', \dots, e_n' Kopien von e, e_1, \dots, e_n sind, in denen alle Vorkommen von $name_1, \dots, name_n$ durch $name_1', \dots, name_n'$ ersetzt werden.

Bedeutung von lokalen Definitionen

- Für die Auswertung:
 - Ersetzen von lokaler durch globale Definition
 - Zugriff auf lokalen Kontext wurde bereits vorher durch Regel (APP) ersetzt ("Substitution")

- (LOCAL) Zugriffe auf formale Parameter in e', e_1', \dots, e_n' wurden bereits durch vorherige Auswertung (APP) ersetzt. Zugriffe auf lokale Definitionen sind möglich, da diese nun in der globalen Umgebung sind.
- $E[(\text{local } name_1, \dots, name_n : e_1, \dots, e_n) \text{ in } e]$
 $\rightarrow (def\ name_1, \dots, name_n = e_1, \dots, e_n \text{ in } e)$
 wobei e, e_1, \dots, e_n nirgendwo im Programm vorkommen
 und e', e_1', \dots, e_n' Kopien von e, e_1, \dots, e_n sind, in denen alle Vorkommen von $name_1, \dots, name_n$ durch $name_1', \dots, name_n'$ ersetzt werden.

Bedeutung von lokalen Definitionen



Bedeutung von lokalen Definitionen

```
(+ 2 (local
  [(define THREE 3)]
  (local
    [(define y (+ THREE 1))]
    (* y 2))))
```

→ (LOCAL)

```
(define THREE_0 3)
```

```
(+ 2 (local
  [(define y (+ THREE_0 1))]
  (* y 2)))
```

→ (PROG)

```
(+ 2 (local
  [(define y (+ THREE_0 1))]
  (* y 2)))
```

→ (LOCAL)

```
(define y_0 (+ THREE_0 1))
```

```
(+ 2 (* y_0 2))
```

→ (PROG, CONST, PRIM)

10

Umgebung:
(define THREE_0 3)

THREE_0 ist in der
Auswertungs-Umgebung

Umgebung:
(define THREE_0 3)
(define y_0 4)

Scoping in Auswertungsregeln

- Semantik: lexikalisches Scoping
 - Gültigkeitsbereich von lokalen Definitionen
 - Nur innerhalb von (local ...) Ausdruck
- Scoping-Regel wird umgesetzt durch zwei Auswertungsregeln
 - (LOCAL)
 - (APP)

Scoping in Auswertungsregeln

- (LOCAL)
 - Umbenennung von lokalen Konstanten
 - Neuer Name darf nirgendwo sonst im Programm definiert sein
 - Innerhalb des (local ...) Ausdrucks
 - Ersetzen des original Namens
 - Durch neuen Namen
- Erzeugter Name ist verschieden von allen benutzerdefinierten Namen
 - Kann nirgendwo zufällig verwendet sein
 - Compiler verbietet Verwendung von Konstanten ohne Definition im Scope
- Einsetzen des erzeugten Namen nur innerhalb des (local ...) Ausdrucks

Scoping in Auswertungsregeln

- (APP)
 - Ähnlich zu (LOCAL):
 - Ersetzung der formalen Parameter im Funktions-Body
 - Ersetzung von “außen” nach “innen”
- Bei Closures als Ergebnis der (APP) Auswertungsregel

- Gebundene Parameter sind bereits ersetzt

- Beispiel

```
(define (f x)  
  (lambda (y) (+ x y)))
```

x ist gebunden an den formalen Parameter.

```
(f 3)
```

```
→ (CONST, APP)
```

```
(lambda (y) (+ 3 y))
```

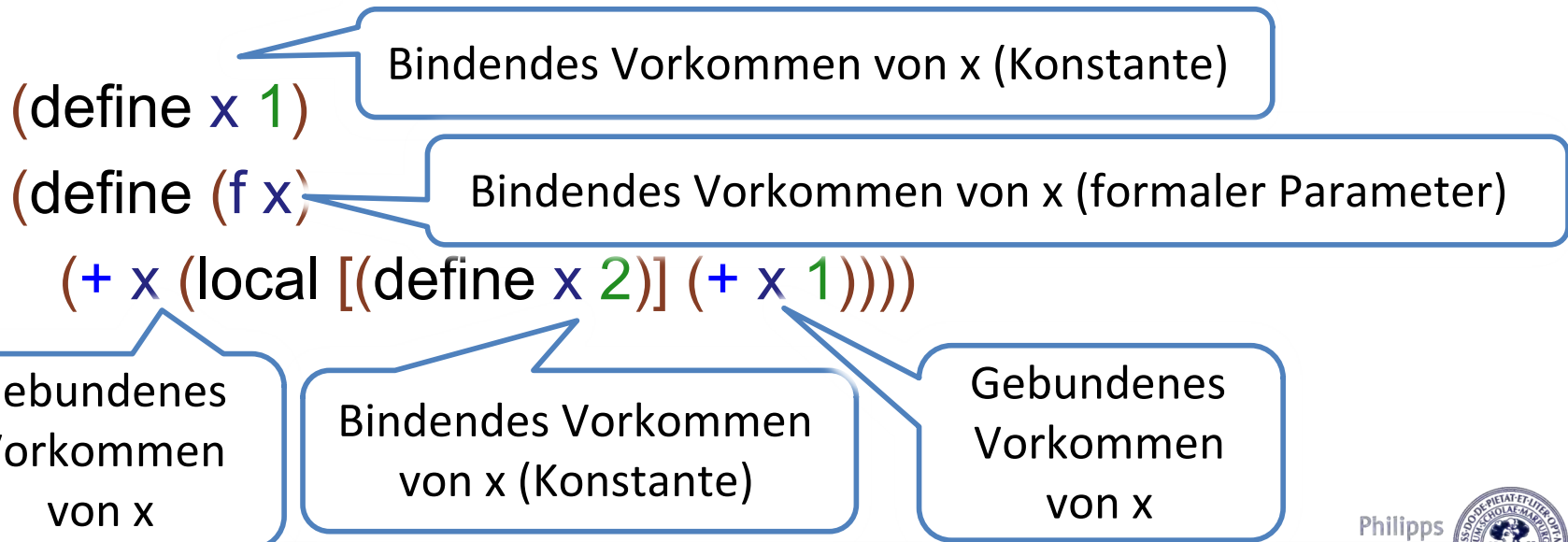
x wird ersetzt durch das Argument.

Shadowing

- Bei Verwendung eines Namens (Konstante oder formaler Parameter)
 - Binden an Definition
 - Es können mehrere Definitionen für den Namen existieren
 - Im aktuellen Scope
 - In den umgebenden Scopes
 - Nur eine Definition pro Scope
 - Bindung immer an die lexikalisch nächste Definition

Shadowing


- Definition einer Konstanten bzw. eines formalen Parameters: “bindendes Vorkommen” des Bezeichners
- Verwendung eines Bezeichners als Ausdruck: “gebundenes Vorkommen”



Shadowing

- Definition einer Konstanten bzw. eines formalen Parameters: “bindendes Vorkommen” des Bezeichners
- Verwendung eines Bezeichners als Ausdruck: “gebundenes Vorkommen”

```
(define x 1)
(define (f x)
  (+ x (local [(define x 2)] (+ x 1))))
> (f 3)
6
```



Shadowing und Modularität

- Modul
 - Unabhängig verstehbare Programmeinheit
 - Bedeutung des Moduls sollte dasselbe sein, unabhängig von dem Kontext in dem es sich befindet
- Beispiel: Ausdruck als Modul
 - Verwendung ungebundener Bezeichner: keine Definition für Bezeichner im aktuellen Scope
 - Verwendung gebundener Bezeichner: Definition für Bezeichner im aktuellen Scope
 - Ausdrücke ohne ungebundene Bezeichner: "geschlossener Term"
- Bedeutung eines geschlossenen Terms sollte unabhängig von der Stelle sein, wo er definiert ist

Scoping Modularität

- Programmiersprachenkonzept: “Blockstruktur”
 - Lokale Definition von Bezeichnern
 - Lexikalisches Scoping
 - Shadowing
- Erste Programmiersprache mit Blockstruktur: ALGOL 60
- Weil eine Blockstruktur gut für das Programmverständnis ist, wird sie von den meisten modernen Programmiersprachen verwendet