Max Wisniewski , Alexander Steen

Tutor: Ansgar Schneider

Aufgabe 1

Ändern Sie die Sprache *WHILE* ab, indem Sie anstelle des atomaren Ausdruckes *read* Anweisungen der Form *read I* zulassen. Die Semantik dieser Anweisung lautet informell: Die Ausführung von *read I* bewirkt eine Zuweisung des nächsten Eingabewertes an die Variable I und eine Verkürzung der Eingabedatei um ein Element.

Formalisieren Sie die Semantik von *read I* denotationell.

Lösung:
Das erste Problem, das uns auffällt, ist das uns die *WHILE* Semantik nicht erlaubt boolsche Werte in Variablen zu speichern. Daher nehmen wir uns entweder die Möglichkeit boolean zu lesen oder aber wir nehmen den Ausdruck nicht anstelle des alten sondern nur zusätzlich.

$$C[readI](s,e,a) := \begin{cases} \underline{Fehler} & , falls\, e \neq ((x.e') \wedge x \in W \\ (s[x/I], e', a) & e = (x.e') \wedge x \in W \end{cases}$$

Sollte nun oben auf der Eingabe kein richtiger Wert liegen oder die Eingabe nicht den richtigen Wert haben wird ein Fehler ausgegeben. Sollte aber ein Wert oben in der Eingabe liegen, wird dieser in das Wörterbuch substituiert.

Aufgabe 2

Erweitern Sie die Sprache *WHILE* um eine Anweisung der Form `FOR I:= T TO N DO C`. Formalisieren Sie die Semantik dieser Form denotationell.

Lösung:

Wir führen eine neue Regel ein, die C so oft ausführt, wie T kleiner als N ist. Als weiteres feature werden wir einen Fehler werfen, falls $T > N$ gilt.

$$C[FOR\, I\; := \; T\; TO\; N\; DO\; C]z := \begin{cases} \underline{Fehler} & , falls\; T > N \\ \underline{Fehler} & , falls C[C]z = \underline{Fehler} \\ z' & , falls T = N \wedge z' = C[C]z \\ C[C; FOR\, I\; := \; T+1\; TON\; DO\; C]]z & , sonst \end{cases}$$

Im ersten Fall sehen wir, dass die Schleife niemals terminieren kann und daher geben wir einen Fehler aus. Im zweiten Fall hat die Ausführung von von C einen Fehler ergeben. Der dritte Fall ist der Anker für die Rekursion, da dort T=N gilt, muss C nur noch einmal ausgeführt werden.
Wenn wir noch mehr zu tun haben, führen wir C einmal aus und fahren danach mit der Schleife fort, wobei T um eins inkrementiert ist.

Aufgabe 3

Erweitern Sie die Sprache *WHILE* um einen atomaren boolschen Term *eof*. Die informelle Semantik von *eof* lautet: *eof* ist wahr gdw. die Eingabe leer ist.
Formalisieren Sie die Semantik von *eof* denotationell.

Lösung:

Der Term *eof* kommt nur in den Boolschen Termen vor, daher muss nur an dieser Stelle eine Veränderung vorgenommen werden.

$$B[eof]z := \begin{cases} (false, z) & , fallsz = (s, \varepsilon, a) \\ (true, z) & , sonst \end{cases}$$

Enthält der Zustand eine leere Eingabe, so gibt *eof* false zurück. Ist dies nicht der Fall so ist der Stack nicht leer und das ganze gibt true zurück.

Aufgabe 4

Programmieren Sie in *WHILE* (einschließlich eof) einen Algorithmus zur Berechnung der Summe aller Eingabewerte. Beweisen Sie die Korrektheit Ihres Programmes anhand der denotationellen Semantik. Diskutieren Sie die Problematik beim Fehlen von *eof*.

Lösung:

Zunächst einmal konkreter Syntax:

```
SUM := 0
WHILE ( not eof )
    X := read
    SUM := SUM + X
output SUM
```

und nun in abstrakter Syntax aus unserem While-Programm

```
allSum :: P
allSum = Seq init (Seq sumPart fin)
sumPart =
    (WHILE eof
        (Seq (Assign X read)
        (Assign SUM (TApp SUM Plus X)))
    )
init = Assign SUM 0
fin = output SUM
```

Nun können wir als nächstes Beweisen, dass $P[allSum](s, [b_1, .., b_n].[], \varepsilon) = (s[SUM/\sum_{k=1}^{n} b_k], [], (\sum_{k=1}^{n} b_k).\varepsilon)$ gilt. Dafür zeigen wir, dass folgendes gilt $s'(SUM) = x \Rightarrow P[sumPart](s', [b_1, .., b_n], a) = (s*, [], a)$, wobei $s*(SUM) = x + \sum_{k=1}^{n} b_i$ ist. per Induktion.

I.A. $n = 0$ (Keine Eingabe) Sei $s'(SUM) = sum$.

$$\begin{aligned} P[allSum](s', \varepsilon, a) &= C[allSum](s', \varepsilon, a) \\ &= (s', \varepsilon, a) \\ &\quad \text{da gilt} \\ B[NOTEOF](s', \varepsilon, a) &= (false, (s', \varepsilon, a)) \end{aligned}$$

Nun gilt das sich der Wert von Sum nicht geändert hat, also $s'(SUM) = x$, und da wir $n = 0$ haben ist $s'(SUM) = x + overset0 \sum_{k=1} b_i$.

I.S. $n \to n+1$
Sei $s'(SUM) = x$, $C' = (Seq(AssignXread)(AssignSUM(TAppSUMPlusX)$ $W' = (While(noteof)C')$

$$
\begin{array}{rl}
& P[allSum](s', [b_{n+1}, b_n, ..., b_1], a) \\
= & C[W'](s', [b_{n+1}, ..., b_1], a) \\
= & C[W'](s', [b_{n+1}, ..., b_1], a) \\
= & C[C'; W'](s', [b_{n+1}, ..., b_1], a)(*) \\
= & C[Seq(AssignSUM(TAppSUMPlusX))W'](C[AssignXread](s', [b_{n+1}, ..., b_1], a)) \\
= & C[Seq(AssignSUM(TAppSUMPlusX))W'](s'[X/b_{n+1}], [b_n, ..., b_1], a) \\
= & C[W'](C[AssignSUM(TappSUMPlusX))](s'[X/b_{n+1}], [b_n, ..., b_1], a))(**) \\
= & C[W'](s'[X/b_{n+1}][SUM/(x + b_{n+1}), [b_n, ..., b_1], a) \\
\overset{I.V.}{=} & (s*, [], a), \text{ mit } s*(SUM) = (x + b_{n+1} + \sum_{k=1}^{n} b_1)
\end{array}
$$

$(*) gilt, da B[NOTEOF](s', [b_{n+1}, ..., b_1], a) = (true, (s', [b_{n+1}, ..., b_1], a))$
$(**)$ gilt da $T[SUMPlusX](s'[X/b_{n+1}, e, a) = (x + b_{n+1}, s'[X/b_{n+1}, e, a)$, da $s'(SUM) = x$ und $s'[X/b_{n+1}(X) = b_{n+1}$.

$\square$

Nun sehen wir, dass durch $C[Seq(AssignSUM0)(SeqW'fin)](s, [b_n, ..., b_1], [])$ $= C[SeqW'fin](s'[SUM/0], [b_n, ..., b_1], [])$ gilt und nach unserer gezeigten Behauptung, wissen wir nun, dass gilt: $C[SeqW'fin](s'[SUM/0], [b_n, ..., b_1], [])$ $= C[fin](s'[SUM/0][SUM/0 + \sum_{k=1}^{n} b_k], [], [])$. Gehen wir also mit output in die letzte anweisung, gilt. $C[outputSUM](s'[SUM/\sum_{k=1}^{n} b_k], [], []) = (s'[SUM/\sum_{k=1}^{n} b_k], [], [\sum_{k=1}^{n} b_k])$
Wir haben gezeigt, dass sowohl in der Ausgabe, als auch im Wörterbuch am Ende der Methode die Summe aller Eingaben stehen wird.

Zum letzten Teil können wir sagen, dass diese Methode nicht ohne *eof* funktionieren kann, da wir uns nicht sicher ein können, wie lange die Eingabe ist. Wir könnten genau so vorgehen, allerdings würde dann im letzten Schritt der Schleife ein Fehler fliegen, da wir mit read auf ein nicht existentes Datum zugreifen.

Eine Möglichkeit wäre es, dass man von der Eingabe verlangt, dass die erste Stelle der Eingabe immer die Länge der gesammten Liste ist. Dies wäre allerdings eine schlechte Abstraktion, da es Diesziplin vom Programmierer bedürfte. Es würde so aber wenigstens laufen. In diesem Fall würde sich die Forschleife aus Aufgabe 3 benutzen.