Auswertungs- und Äquivalenzregeln der BSL

Auswertungsregeln

(KONG) $E[e_1] \to E[e_2]$ falls $e_1 \to e_2$
Skript 8.6

‹E› ::= []
| (‹name› ‹v›* ‹E› ‹e›*)
| (**cond** [‹E› ‹e›] { [‹e› ‹e›]}*)
| (**and** ‹E› ‹e›+)
| (**and true** ‹E›)

(PROG) *Skript 8.8* 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 in der erweiterten Umgebung fortgesetzt. Ist das nächste Programmelement ein Ausdruck, so wird dieser gemäß der unten stehenden Regeln in der aktuellen Umgebung zu einem Wert ausgewert. Ist das nächste Programmelement 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.

(FUN) (*name* v_1 … v_n) → *e*[*name*$_1$:= v_1 … *name*$_n$:= v_n] falls (**define** (*name name*$_1$ … *name*$_n$) *e*) in Umgebung
Skript 8.9.1

(PRIM) (*name* v_1 … v_n) → *v* falls *name* eine primitive Funktion *f* ist und $f(v_1, …, v_n) = v$
Skript 8.9.1

(CONST) *name* → *v* falls (**define** *name v*) in Umgebung
Skript 8.9.2

(COND-True) (**cond** [**true** *e*] …) → *e*
Skript 8.9.3

(COND-False) (**cond** [**false** e_1] [e_2 e_3] …) → (**cond** [e_2 e_3] …)
Skript 8.9.3

(AND-1) (**and true true**) → **true**
Skript 8.9.4

(AND-2) (**and true false**) → **false**
Skript 8.9.4

(AND-3) (**and false** …) → **false**
Skript 8.9.4

(AND-4) (**and true** e_1 e_2 …) → (**and** e_1 e_2 …)
Skript 8.9.4

(STRUCT-make) (**make-***name* v_1 … v_n) → <**make-***name* v_1 … v_n> falls (**define-struct** *name* (*name*$_1$ … *name*$_n$)) in Umgebung
Skript 8.9.5

(STRUCT-select) (name-*name*$_i$ <***make-****name* v_1 … v_n>) → v_i falls (**define-struct** *name* (*name*$_1$ … *name*$_n$)) in Umgebung
Skript 8.9.5

(STRUCT-predtrue) (*name***?** <**make-***name* …>) → **true**
Skript 8.9.5

(STRUCT-predfalse) (*name***?** *v*) → **false** falls v nicht <**make-***name* …> ist
Skript 8.9.5

Äquivalenzregeln

(EKONG) $E[e_1] \equiv E[e_2]$ falls $e_1 \equiv e_2$
Skript 8.12

‹E› ::= []
| (‹name› ‹e›* ‹E› ‹e›*)
| (**cond** { [‹e› ‹e›]}* [‹E› ‹e›] { [‹e› ‹e›]}*)
| (**cond** { [‹e› ‹e›]}* [‹e› ‹E›] { [‹e› ‹e›]}*)
| (**and** ‹e›* ‹E› ‹e›*)

(EPRIM) Äquivalenzen primitiver Funktionen, Bsp: *(+ a b) ≡ (+ b a)*
Skript 8.12

(EREFL) $e \equiv e$
Skript 8.12

(EKOMM) $e_2 \equiv e1$ falls $e_1 \equiv e_2$
Skript 8.12

(ETRANS) $e_1 \equiv e_3$ falls $e_1 \equiv e_2$ und $e_2 \equiv e_3$
Skript 8.12

(ERED) $e_1 \equiv e_2$ falls $e_1 \to e_2$
Skript 8.12

(EFUN) (*name* e_1 … e_n) ≡ *e*[*name*$_1$:= e_1 … *name*$_n$:= e_n] falls (**define** (*name name*$_1$ … *name*$_n$) *e*) in Umgebung
Skript 8.12

(EAND) (**and** … **false** …) ≡ **false**
Skript 8.12

Auswertungsregeln der ISL

(KONG) $E[e_1] \rightarrow E[e_2]$ falls $e_1 \rightarrow e_2$

Skript 12.4

‹E› ::= []
| (‹v›* ‹E› ‹e›*)

(PROG) Ein Programm wird von links nach rechts ausgeführt und startet mit der leeren Umgebung. Ist das nächste Programmelement ein Ausdruck, so wird dieser gemäß der unten stehenden Regeln in der aktuellen Umgebung zu einem Wert ausgewert. Ist das nächste Programmelement 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.

Skript 12.3

(APP) ((lambda ($name_1$ ... $name_n$) *e*) v_1 ... v_n) → *e*[$name_1$:= v_1 ... $name_n$:= v_n]

Skript 12.5.1

(PRIM) (*v* v_1 ... v_n) → *v'* falls v eine primitive Funktion *f* ist und $f(v_1, ..., v_n) = v'$

Skript 12.5.1

(CONST) *name* → *v* falls (**define** *name v*) in Umgebung

Skript 12.5.3

(LOCAL) *E[(local [(define* $name_1$ e_1 *) ... (define* $name_n$ e_n *)] e)] →*

Skript 12.5.2

(define $name_1'$ e_1' *) ... (define* $name_n'$ e_n' *) E[e']*

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