

5.6 Getypte Variablen While_T

Bisher speicherten alle Variablen in unseren Programmen ausschließlich ganze Zahlen, aber keine booleschen Werte. Wir ändern While_B jetzt, sodass Zuweisungen auch für boolesche Werte erlaubt sind, und nennen die neue Sprache While_T . Dabei stößt man aber schnell auf ein Problem: Welchen Wert hat y am Ende des Programms

$$x := \text{true}; y := x + 5?$$

Genau genommen möchten wir dieses Programm als ungültig zurückweisen. Im Allgemeinen ist es aber sehr schwierig (bzw. unentscheidbar), ob ein Programm in diesem Sinn als gültig anzusehen ist.

Übung: Welche der folgenden Programme sollten Ihrer Meinung nach als ungültig zurückgewiesen werden?

- $x := \text{true}; x := 0$
- $y := \text{true}; (\text{if } (y) \text{ then } x := \text{true} \text{ else } x := 0); z := x$
- $x := \text{true}; \{ \text{var } x = 0; y := x \}; y := y + 2$

Typen legen die möglichen Wertebereiche für Variablen (und damit auch Ausdrücke) fest, in unserem Fall ganze Zahlen bzw. Wahrheitswerte. Ein (statisches) *Typsystem* ist eine spezielle Form der Programmanalyse bzw. -verifikation, die die Typen eines Programms „zur Übersetzungszeit“ automatisch (und effizient) analysiert. Dabei wird das Programm insbesondere *nicht* ausgeführt. Das Typsystem garantiert, dass bestimmte Laufzeitfehler in typkorrekten Programmen nicht auftreten können, beispielsweise, dass die Operanden einer arithmetischen Operation wirklich Zahlen (und keine Wahrheitswerte) sind. Viele Programmier- und Tippfehler zeigen sich bereits durch Typfehler: „Well-typed programs cannot go wrong“ [R. Milner]. Aber nicht alle Laufzeitfehler können mittels Typsysteme ausgeschlossen werden, z.B. Division durch 0.

5.6.1 Typen für While_T

Wir unterscheiden ab jetzt arithmetische und boolesche Ausdrücke nicht mehr syntaktisch, das Typsystem wird sich später um die Trennung kümmern.

Definition 19 (Ungetypte Ausdrücke). Die Syntax für (ungetypte) Ausdrücke Exp lautet:

$$\text{Exp } e ::= n \mid x \mid e_1 - e_2 \mid e_1 * e_2 \mid \text{true} \mid e_1 \leq e_2 \mid \text{not } b \mid b_1 \ \&\& \ b_2$$

Entsprechend passt sich auch die Syntax für Anweisungen an: Wo bisher arithmetische oder boolesche Ausdrücke gefordert waren, verlangen wir nur noch (ungetypte) Ausdrücke:

Definition 20 (Syntax für Anweisungen in While_T).

$$\text{Com } c ::= \text{skip} \mid x := e \mid c_1; c_2 \mid \text{if } (e) \text{ then } c_1 \text{ else } c_2 \mid \text{while } (e) \text{ do } c \mid \{ \text{var } x = e; c \}$$

Definition 21 (Typen). int und bool sind die Typen \mathbb{T} für While_T :

$$\mathbb{T} = \{\text{int}, \text{bool}\}$$

Nach Variablenkonvention steht die Metavariablen τ immer für einen Typen aus \mathbb{T} .

Die Annahme für unser statisches Typsystem lautet: Jede Variable hat (innerhalb ihres Gültigkeitsbereichs) einen festen Typ. Damit brauchen wir uns keine Gedanken darüber zu machen, ob eine Variable an Kontrollzusammenflussstellen (z.B. nach einer Fallunterscheidung) immer den gleichen Typ hat, egal wie man an diese Stelle gelangt ist.

Für die Typkorrektheitsüberprüfung gibt es drei Ansätze:

1. Die Sprachdefinition legt fest, welche Variablen für Zahlen und welche für Wahrheitswerte verwendet werden dürfen. Dies kommt jedoch wieder einer syntaktischen Trennung arithmetischer und boolescher Ausdrücke gleich und in keiner modernen Programmiersprache mehr vor. In Fortran haben standardmäßig alle Variablen mit den Anfangsbuchstaben i, j, k, l, m oder n den Typ Integer und alle anderen den Typ Real.
2. Jedes Programm deklariert die Typen der Variablen, die es verwendet. Das Typsystem führt also lediglich eine *Typüberprüfung* durch.
3. Der Typ wird aus den Verwendungen der Variablen erschlossen. Das Typsystem führt eine *Typinferenz* durch. Bei Typinferenz tritt in der Praxis, insbesondere wenn Prozeduren und Funktionen involviert sind, oft das Problem auf, dass Fehlermeldungen des Typcheckers nur schwer zu verstehen sind, weil der Typfehler erst an einer ganz anderen Stelle auftritt, als der eigentliche Tippfehler des Programmierers ist.

Beispiel 17. Für die Anweisung `b := c; i := 42 + j` sehen könnten diese drei Ansätze wie folgt aussehen:

1. Die Sprachdefinition legt fest, dass Variablen, deren Namen mit `i` und `j` beginnen, in *allen* Programmen immer den Typ `int` haben, und dass alle anderen Variablen nur Wahrheitswerte speichern.
2. Das Programm selbst deklariert (in einer geeigneten Notation), dass `i` und `j` vom Typ `int` sind, und dass `b` und `c` vom Typ `bool` (oder auch `int`) sind.
3. Das Typsystem erschließt die Typen aus der Verwendung der Variablen. Da `j` als Operand eines `+` vorkommt, muss `j` den Typ `int` haben. Da der Variablen `i` das Ergebnis einer Addition zugewiesen wird, muss auch sie den Typ `int` haben. Für die Variablen `b` und `c` lässt sich nur bestimmen, dass beide den gleichen Typ haben müssen, nicht aber, ob dies nun `int` oder `bool` sein soll.

Im Folgenden gehen wir davon aus, dass ein Programm auch die Typen der verwendeten (globalen) Variablen deklariert.

Definition 22 (Typkontext). Ein *Typkontext* $\Gamma :: \text{Var} \Rightarrow \mathbb{T}$ ordnet jeder Variablen einen Typ zu.

Ein Programm besteht also aus der auszuführenden Anweisung `c` und einem Typkontext Γ .

Beispiel 18. Für die Anweisung

$$c \equiv \text{while } (x \leq 50) \text{ do } (\text{if } (y) \text{ then } x := x * 2 \text{ else } x := x + 5)$$

ist $\Gamma \equiv [x \mapsto \text{int}, y \mapsto \text{bool}]$ ein „passender“ Typkontext. In diesem Kontext hat die Variable `x` den Typ `int`, `y` den Typ `bool`.

5.6.2 Ein Typsystem für While_T

Ein Typsystem wird wie eine Semantik durch ein Regelsystem für eine Relation $_ \vdash _ :: _$ definiert. Dabei bedeutet $\Gamma \vdash e :: \tau$, dass im Kontext Γ der Ausdruck `e` den Typ τ hat.

Definition 23 (Typregeln für Ausdrücke). Die Typregeln für Ausdrücke Exp sind im Einzelnen:

$$\begin{array}{c}
\text{TNUM: } \Gamma \vdash n :: \text{int} \quad \text{TVAR: } \frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau} \quad \text{TTRUE: } \Gamma \vdash \text{true} :: \text{bool} \\
\text{TMINUS: } \frac{\Gamma \vdash e_1 :: \text{int} \quad \Gamma \vdash e_2 :: \text{int}}{\Gamma \vdash e_1 - e_2 :: \text{int}} \quad \text{TTIMES: } \frac{\Gamma \vdash e_1 :: \text{int} \quad \Gamma \vdash e_2 :: \text{int}}{\Gamma \vdash e_1 * e_2 :: \text{int}} \\
\text{TLEQ: } \frac{\Gamma \vdash e_1 :: \text{int} \quad \Gamma \vdash e_2 :: \text{int}}{\Gamma \vdash e_1 \leq e_2 :: \text{bool}} \quad \text{TNOT: } \frac{\Gamma \vdash e :: \text{bool}}{\Gamma \vdash \text{not } e :: \text{bool}} \quad \text{TAND: } \frac{\Gamma \vdash e_1 :: \text{bool} \quad \Gamma \vdash e_2 :: \text{bool}}{\Gamma \vdash e_1 \ \&\& \ e_2 :: \text{bool}}
\end{array}$$

Beispiel 19. Sei $\Gamma \equiv [x \mapsto \text{int}, y \mapsto \text{bool}]$. Dann gilt $\Gamma \vdash (x \leq 10) \ \&\& \ (\text{not } y) :: \text{bool}$, aber es gibt kein τ , für das $\Gamma \vdash y \leq x :: \tau$ gelte. Die Typaussage $\Gamma \vdash (x \leq 10) \ \&\& \ (\text{not } y) :: \tau$ wird wie bei der Semantik durch einen Ableitungsbaum hergeleitet:

$$\frac{\frac{\frac{\Gamma(x) = \text{int}}{\Gamma \vdash x :: \text{int}} \text{TVAR} \quad \frac{}{\Gamma \vdash 10 :: \text{int}} \text{TNUM}}{\Gamma \vdash x \leq 10 :: \text{bool}} \text{TLEQ} \quad \frac{\frac{\Gamma(y) = \text{bool}}{\Gamma \vdash y :: \text{bool}} \text{TVAR}}{\Gamma \vdash \text{not } y :: \text{bool}} \text{TNOT}}{\Gamma \vdash (x \leq 10) \ \&\& \ (\text{not } y) :: \text{bool}} \text{TAND}$$

Definition 24 (Typkorrektheit von Ausdrücken). Ein Ausdruck e heißt *typkorrekt* in einem Typkontext Γ , falls e einen Typ hat, d.h., falls es ein τ gibt, sodass $\Gamma \vdash e :: \tau$.

Definition 25 (Typkorrektheit von Anweisungen). Anweisungen selbst haben keinen Typ, müssen die Ausdrücke aber korrekt verwenden. Typkorrektheit $\Gamma \vdash c \checkmark$ für eine Anweisung c im Typkontext Γ ist auch durch ein Regelsystem definiert:

$$\begin{array}{c}
\text{TSKIP: } \Gamma \vdash \text{skip} \checkmark \quad \text{TASS: } \frac{\Gamma(x) = \tau \quad \Gamma \vdash e :: \tau}{\Gamma \vdash x := e \checkmark} \quad \text{TSEQ: } \frac{\Gamma \vdash c_1 \checkmark \quad \Gamma \vdash c_2 \checkmark}{\Gamma \vdash c_1; c_2 \checkmark} \\
\text{TIF: } \frac{\Gamma \vdash e :: \text{bool} \quad \Gamma \vdash c_1 \checkmark \quad \Gamma \vdash c_2 \checkmark}{\Gamma \vdash \text{if } (e) \ \text{then } c_1 \ \text{else } c_2 \checkmark} \quad \text{TWHILE: } \frac{\Gamma \vdash e :: \text{bool} \quad \Gamma \vdash c \checkmark}{\Gamma \vdash \text{while } (e) \ \text{do } c \checkmark} \\
\text{TBLOCK: } \frac{\Gamma \vdash e :: \tau \quad \Gamma[x \mapsto \tau] \vdash c \checkmark}{\Gamma \vdash \{ \text{var } x = e; c \} \checkmark}
\end{array}$$

Definition 26 (typisierbar, typkorrekte Programme). Eine Anweisung c heißt *typisierbar*, wenn es einen Kontext Γ mit $\Gamma \vdash c \checkmark$. Ein Programm $P \equiv (c, \Gamma)$ heißt *typkorrekt*, falls $\Gamma \vdash c \checkmark$.

Die Regel TBLOCK für Blöcke ist keine reine Typüberprüfung mehr, sondern inferiert den (neuen) Typ für x aus dem Typ des Initialisierungsausdrucks e . Wollte man eine reine Typüberprüfung, müsste ein Block entweder den neuen Typ für x deklarieren (z.B. $\{ \text{int } x = 5; c \}$) oder der Typ für x dürfte sich nicht ändern.

Bemerkung: Die Typsystem-Regeln für $_ \vdash _ :: _$ und $_ \vdash _ \checkmark$ spezifizieren bereits einen (terminierenden) Algorithmus, da die Annahmen der Regeln stets kleinere Ausdrücke oder Anweisungen enthalten als die Konklusion.

Übung: Welche der Programme vom Beginn des Abschnitts 5.6 sind typkorrekt – bei geeigneter Typdeklaration der globalen Variablen?

5.6.3 Small-Step-Semantik für While_T

Zustände müssen jetzt neben Zahlen auch Wahrheitswerte speichern können:

$$\Sigma \equiv \text{Var} \Rightarrow \mathbb{Z} + \mathbb{B}$$

Bei der Auswertung eines Ausdrucks e in einem Zustand σ kann es nun passieren, dass die Werte in σ nicht zu den erwarteten Typen in e passen. Deswegen kann auch unsere neue Auswertungsfunktion $\mathcal{E} \llbracket e \rrbracket \sigma$ für den Ausdruck e im Zustand σ nur noch partiell sein:

$$\mathcal{E} \llbracket _ \rrbracket _ :: \text{Exp} \Rightarrow \Sigma \rightarrow \mathbb{Z} + \mathbb{B}$$

Wenn $\mathcal{E} \llbracket e \rrbracket \sigma$ nicht definiert ist, schreiben wir $\mathcal{E} \llbracket e \rrbracket \sigma = \perp$. Umgekehrt bedeutet $\mathcal{E} \llbracket e \rrbracket \sigma = v$ (wobei $v \in \mathbb{Z} + \mathbb{B}$), dass $\mathcal{E} \llbracket e \rrbracket \sigma$ definiert ist und den Wert v hat.

Definition 27 (Auswertungsfunktion für Ausdrücke). Die Auswertungsfunktion $\mathcal{E} \llbracket e \rrbracket \sigma$ kombiniert lediglich die bisherigen Auswertungsfunktionen $\mathcal{A} \llbracket _ \rrbracket _$ und $\mathcal{B} \llbracket _ \rrbracket _$:

$$\begin{array}{l|l} \mathcal{E} \llbracket n \rrbracket \sigma & = \mathcal{N} \llbracket n \rrbracket \\ \mathcal{E} \llbracket x \rrbracket \sigma & = \sigma(x) \\ \mathcal{E} \llbracket e_1 - e_2 \rrbracket \sigma & = \mathcal{E} \llbracket e_1 \rrbracket \sigma - \mathcal{E} \llbracket e_2 \rrbracket \sigma \\ \mathcal{E} \llbracket e_1 * e_2 \rrbracket \sigma & = \mathcal{E} \llbracket e_1 \rrbracket \sigma \cdot \mathcal{E} \llbracket e_2 \rrbracket \sigma \end{array} \quad \left| \quad \begin{array}{l} \mathcal{E} \llbracket \text{true} \rrbracket \sigma & = \text{tt} \\ \mathcal{E} \llbracket e_1 \leq e_2 \rrbracket \sigma & = \mathcal{E} \llbracket e_1 \rrbracket \sigma \leq \mathcal{E} \llbracket e_2 \rrbracket \sigma \\ \mathcal{E} \llbracket \text{not } e \rrbracket \sigma & = \neg \mathcal{E} \llbracket e \rrbracket \sigma \\ \mathcal{E} \llbracket e_1 \ \&\& \ e_2 \rrbracket \sigma & = \mathcal{E} \llbracket e_1 \rrbracket \sigma \wedge \mathcal{E} \llbracket e_2 \rrbracket \sigma \end{array}$$

Dabei sind die Operatoren $-$, \cdot , \leq , \neg und \wedge auch als partielle Funktionen auf dem Wertebereich $\mathbb{Z} + \mathbb{B}$ zu verstehen: Sie sind nur auf ihrem bisherigen Wertebereich (\mathbb{Z} oder \mathbb{B}) definiert und für alle anderen Werten aus $\mathbb{Z} + \mathbb{B}$ undefiniert. Genauso sind sie undefiniert, wenn eines ihrer Argumente undefiniert ist.

Beispiel 20. $\mathcal{E} \llbracket \text{false} \ \&\& \ (1 \leq \text{true}) \rrbracket \sigma = \perp$, weil $1 \leq \text{tt} = \perp$ und damit auch $\text{ff} \wedge \perp = \perp$.

Definition 28 (Small-Step-Semantik). Die Small-Step-Semantik für While_T besteht nun aus folgenden Regeln:

$$\begin{array}{c} \text{ASS}_{\text{SS}}^{\text{T}}: \frac{\mathcal{E} \llbracket e \rrbracket \sigma = v}{\langle x := e, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma[x \mapsto v] \rangle} \\ \\ \text{SEQ1}_{\text{SS}}^{\text{T}}: \frac{\langle c_0, \sigma \rangle \rightarrow_1 \langle c'_0, \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \rightarrow_1 \langle c'_0; c_1, \sigma' \rangle} \quad \text{SEQ2}_{\text{SS}}^{\text{T}}: \langle \text{skip}; c, \sigma \rangle \rightarrow_1 \langle c, \sigma \rangle \\ \\ \text{IFTT}_{\text{SS}}^{\text{T}}: \frac{\mathcal{E} \llbracket e \rrbracket \sigma = \text{tt}}{\langle \text{if } (e) \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_1 \langle c_0, \sigma \rangle} \\ \\ \text{IFFF}_{\text{SS}}^{\text{T}}: \frac{\mathcal{E} \llbracket e \rrbracket \sigma = \text{ff}}{\langle \text{if } (e) \text{ then } c_0 \text{ else } c_1, \sigma \rangle \rightarrow_1 \langle c_1, \sigma \rangle} \\ \\ \text{WHILE}_{\text{SS}}^{\text{T}}: \langle \text{while } (e) \text{ do } c, \sigma \rangle \rightarrow_1 \langle \text{if } (e) \text{ then } c; \text{ while } (e) \text{ do } c \text{ else skip}, \sigma \rangle \\ \\ \text{BLOCK1}_{\text{SS}}^{\text{T}}: \frac{\mathcal{E} \llbracket e \rrbracket \sigma = v \quad \langle c, \sigma[x \mapsto v] \rangle \rightarrow_1 \langle c', \sigma' \rangle}{\langle \{ \text{var } x = e; c \}, \sigma \rangle \rightarrow_1 \langle \{ \text{var } x = \mathcal{V}^{-1} \llbracket \sigma'(x) \rrbracket; c' \}, \sigma'[x \mapsto \sigma(x)] \rangle} \\ \\ \text{BLOCK2}_{\text{SS}}^{\text{T}}: \frac{\mathcal{E} \llbracket e \rrbracket \sigma \neq \perp}{\langle \{ \text{var } x = e; \text{skip} \}, \sigma \rangle \rightarrow_1 \langle \text{skip}, \sigma \rangle} \end{array}$$

Dabei ist $\mathcal{V}^{-1} \llbracket v \rrbracket$ die Umkehrung von $\mathcal{E} \llbracket _ \rrbracket _$ für Werte $v \in \mathbb{Z} + \mathbb{B}$, also die Verallgemeinerung von

$\mathcal{N}^{-1} \llbracket _ \rrbracket$ auf den neuen Wertebereich für Variablen. Regel BLOCK2_{SS}^T ist nur anwendbar, wenn der Initialisierungsausdruck e keine Typfehler bei der Auswertung hervorruft.

Übung: Zeigen Sie, dass es neben $\langle \text{skip}, \sigma \rangle$ weitere blockierte Konfigurationen in dieser Small-Step-Semantik gibt. Woran liegt das?

5.6.4 Typsicherheit von While_T

Die Semantik ist unabhängig vom Typsystem und seinen Regeln. Semantik beschreibt das *dynamische* Verhalten eines Programms, das Typsystem eine *statische* Programmanalyse. Wie alle Programm-Analysen gibt ein Typsystem ein Korrektheitsversprechen, das mittels der Semantik bewiesen werden kann.

Definition 29 (Korrektheit, Typsicherheit, Vollständigkeit). Ein Typsystem ist *korrekt*, falls es zur Laufzeit keine Typfehler gibt. Laufzeit-Typfehler äußern sich in blockierten Konfigurationen der Small-Step-Semantik, die keine Endzustände (**skip**) sind. Sprachen mit einem korrekten Typsystem heißen *typsicher*. Dual zur Korrektheit ist der Begriff der Vollständigkeit: Das Typsystem ist *vollständig*, wenn jedes Programm ohne Laufzeittypfehler typkorrekt ist.

Korrektheit ist wünschenswert, Vollständigkeit im Allgemeinen (für korrekte Typsysteme) unerreichbar, weil dann das Typsystem nicht mehr entscheidbar sein kann. Viele moderne Programmiersprachen sind (nachgewiesenermaßen) typsicher: ML, Java, C#, Haskell. Nicht typsicher sind beispielsweise Eiffel, C, C++ und Pascal.

Beispiel 21. Das Typsystem für While_T ist nicht vollständig. Das Programm $x := 0; x := \text{true}$ ist nicht typkorrekt, verursacht aber trotzdem keine Laufzeittypfehler.

Um die Typsicherheit von While_T formulieren zu können, brauchen wir noch zwei Begriffe: Den Typ eines Wertes und Zustandskonformanz.

Definition 30 (Typ eines Wertes). Der *Typ* $\text{type}(v)$ eines Wertes $v \in \mathbb{Z} + \mathbb{B}$ ist:

$$\text{type}(v) \equiv \begin{cases} \text{int} & \text{falls } v \in \mathbb{Z} \\ \text{bool} & \text{falls } v \in \mathbb{B} \end{cases}$$

Definition 31 (Zustandskonformanz). Ein Zustand σ ist zu einem Typkontext Γ *konformant*, notiert als $\sigma :: \Gamma$, wenn $\text{type}(\sigma(x)) = \Gamma(x)$ für alle $x \in \text{Var}$.

Typsicherheit für eine Small-Step-Semantik besteht klassischerweise aus zwei Teilen: Fortschritt und Typerhaltung (progress und preservation) nach Wright und Felleisen.

Theorem 21 (Typsicherheit). Sei $\Gamma \vdash c \checkmark$ und $\sigma :: \Gamma$.

Progress Wenn $c \neq \text{skip}$, dann gibt es c' und σ' mit $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$.

Preservation Wenn $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$, dann $\Gamma \vdash c' \checkmark$ und $\sigma' :: \Gamma$.

Den Beweis teilen wir auf die folgenden Hilfslemmata auf:

Lemma 22 (Typkorrektheit von $\mathcal{E} \llbracket _ \rrbracket _$).

Wenn $\Gamma \vdash e :: \tau$ und $\sigma :: \Gamma$, dann ist $\mathcal{E} \llbracket e \rrbracket \sigma$ definiert und $\text{type}(\mathcal{E} \llbracket e \rrbracket \sigma) = \tau$.

Beweis. Regel-Induktion über $\Gamma \vdash e :: \tau$.

- Fälle TNUM, TTRUE: Trivial.
- Fall TVAR: Definiertheit ist trivial. Wegen Zustandskonformanz $\sigma :: \Gamma$ gilt:
 $\text{type}(\mathcal{E} \llbracket e \rrbracket \sigma) = \text{type}(\sigma(x)) = \Gamma(x) = \tau$.
- Fall TMINUS:
 Induktionsannahmen: $\mathcal{E} \llbracket e_1 \rrbracket \sigma$ und $\mathcal{E} \llbracket e_2 \rrbracket \sigma$ sind definiert mit $\text{type}(\mathcal{E} \llbracket e_1 \rrbracket \sigma) = \text{type}(\mathcal{E} \llbracket e_2 \rrbracket \sigma) = \text{int}$.
 Nach Definition von $\text{type}(_)$ ist somit $\mathcal{E} \llbracket e_1 \rrbracket \sigma \in \mathbb{Z}$ und $\mathcal{E} \llbracket e_2 \rrbracket \sigma \in \mathbb{Z}$. Also ist auch $\mathcal{E} \llbracket e_1 - e_2 \rrbracket \sigma$
 definiert mit $\mathcal{E} \llbracket e_1 - e_2 \rrbracket \sigma \in \mathbb{Z}$, also $\text{type}(\mathcal{E} \llbracket e_1 - e_2 \rrbracket \sigma) = \text{int}$.
- Fälle TTIMES, TLEQ, TNOT, TAND: Analog. □

Lemma 23 (Fortschritt).

Wenn $\Gamma \vdash c \checkmark$ und $\sigma :: \Gamma$ mit $c \neq \text{skip}$, dann gibt es c' und σ' mit $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$.

Beweis. Induktion über $\Gamma \vdash c \checkmark$ oder c direkt – analog zu Lem. 3 (σ beliebig). Für die Definiertheit von $\mathcal{E} \llbracket e \rrbracket \sigma$ in den Regelannahmen verwendet man Lem. 22. Bei Blöcken ist wie bei der Sequenz eine Fallunterscheidung über $c = \text{skip}$ nötig, für den induktiven Fall $c \neq \text{skip}$ braucht man die Induktionshypothese mit $\sigma[x \mapsto v]$ für σ . □

Lemma 24 (Erhalt der Zustandskonformanz).

Wenn $\Gamma \vdash c \checkmark$, $\sigma :: \Gamma$ und $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$, dann $\sigma' :: \Gamma$.

Beweis. Regel-Induktion über $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$ (Γ beliebig).

- Fall ASS_{SS}^T: Zu zeigen: Wenn $\Gamma \vdash x := e \checkmark$, $\sigma :: \Gamma$ und $\mathcal{E} \llbracket e \rrbracket \sigma = v$, dann $\sigma[x \mapsto v] :: \Gamma$.
 Aus $\Gamma \vdash x := e \checkmark$ erhält man mit Regelinversion (TASS) τ mit $\Gamma(x) = \tau$ und $\Gamma \vdash e :: \tau$. Aus $\Gamma \vdash e :: \tau$ und $\sigma :: \Gamma$ folgt nach Lem. 22, dass $\text{type}(v) = \tau$. Zusammen mit $\sigma :: \Gamma$ und $\Gamma(x) = \tau$ folgt die Behauptung $\sigma[x \mapsto v] :: \Gamma$.
- Fall SEQ1_{SS}^T: Fall-Annahmen: $\langle c_1, \sigma \rangle \rightarrow_1 \langle c'_1, \sigma' \rangle$, $\Gamma \vdash c_1$; $c_2 \checkmark$, $\sigma :: \Gamma$.
 Induktionsannahme: Für alle Γ gilt: Wenn $\Gamma \vdash c_1 \checkmark$ und $\sigma :: \Gamma$, dann $\sigma' :: \Gamma$.
 Zu zeigen: $\sigma' :: \Gamma$.
 Aus $\Gamma \vdash c_1$; $c_2 \checkmark$ erhält man durch Regelinversion (TSEQ), dass $\Gamma \vdash c_1 \checkmark$ und $\Gamma \vdash c_2 \checkmark$. Mit $\sigma :: \Gamma$ folgt die Behauptung aus der Induktionsannahme.
- Fälle SEQ2_{SS}^T, IF_{TT}_{SS}^T, IFFF_{SS}^T, WHILE_{SS}^T, BLOCK2_{SS}^T: Trivial, da $\sigma' = \sigma$.
- Fall BLOCK1_{SS}^T:
 Fall-Annahmen: $\mathcal{E} \llbracket e \rrbracket \sigma = v$, $\langle c, \sigma[x \mapsto v] \rangle \rightarrow_1 \langle c', \sigma' \rangle$, $\Gamma \vdash \{ \text{var } x = e; c \} \checkmark$ und $\sigma :: \Gamma$.
 Induktionsannahme: Für beliebige Γ gilt: Wenn $\Gamma \vdash c \checkmark$ und $\sigma[x \mapsto v] :: \Gamma$, dann $\sigma' :: \Gamma$.
 Zu zeigen: $\sigma'[x \mapsto \sigma(x)] :: \Gamma$.
 Aus $\Gamma \vdash \{ \text{var } x = e; c \} \checkmark$ erhält man durch Regelinversion (TBLOCK) τ mit $\Gamma \vdash e :: \tau$ und $\Gamma[x \mapsto \tau] \vdash c \checkmark$. Aus $\Gamma \vdash e :: \tau$ und $\sigma :: \Gamma$ folgt mit Lem. 22, dass $\text{type}(v) = \tau$. Zusammen mit $\sigma :: \Gamma$ gilt damit auch $\sigma[x \mapsto v] :: \Gamma[x \mapsto \tau]$. Aus der Induktionsannahme mit $\Gamma[x \mapsto \tau]$ für Γ erhält man damit $\sigma' :: \Gamma[x \mapsto \tau]$. Wegen $\sigma :: \Gamma$ ist $\Gamma(x) = \text{type}(\sigma(x))$. Damit gilt auch die Behauptung $\sigma'[x \mapsto \sigma(x)] :: \Gamma$. □

Lemma 25 (Subject reduction). Wenn $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$, $\Gamma \vdash c \checkmark$ und $\sigma :: \Gamma$, dann $\Gamma \vdash c' \checkmark$.

Beweis. Regel-Induktion über $\langle c, \sigma \rangle \rightarrow_1 \langle c', \sigma' \rangle$ (Γ beliebig).

- Fälle ASS_{SS}^T, BLOCK2_{SS}^T: Trivial mit Regel TSKIP.
- Fall SEQ1_{SS}^T:
 Induktionsannahme: Wenn $\Gamma \vdash c_1 \checkmark$ und $\sigma :: \Gamma$, dann $\Gamma \vdash c'_1 \checkmark$.
 Zu zeigen: Wenn $\Gamma \vdash c_1$; $c_2 \checkmark$ und $\sigma :: \Gamma$, dann $\Gamma \vdash c'_1$; $c_2 \checkmark$.
 Aus $\Gamma \vdash c_1$; $c_2 \checkmark$ erhält man durch Regelinversion (TSEQ), dass $\Gamma \vdash c_1 \checkmark$ und $\Gamma \vdash c_2 \checkmark$. Mit $\sigma :: \Gamma$ folgt aus der Induktionsannahme, dass $\Gamma \vdash c'_1 \checkmark$. Zusammen mit $\Gamma \vdash c_2 \checkmark$ folgt die Behauptung mit Regel TSEQ.

- Fälle SEQ2_{SS}^T , IFTT_{SS}^T , IFFF_{SS}^T : Trivial mit Regelinversion (TSEQ bzw. TIF).
- Fall WHILE_{SS}^T : Aus $\Gamma \vdash \text{while } (e) \text{ do } c \checkmark$ folgt mit Regelinversion, dass $\Gamma \vdash e :: \text{bool}$ und $\Gamma \vdash c \checkmark$. Damit gilt:

$$\frac{\Gamma \vdash e :: \text{bool} \quad \frac{\Gamma \vdash c \checkmark \quad \Gamma \vdash \text{while } (e) \text{ do } c \checkmark}{\Gamma \vdash c; \text{while } (e) \text{ do } c \checkmark} \text{TSEQ} \quad \frac{}{\Gamma \vdash \text{skip} \checkmark} \text{TSKIP}}{\Gamma \vdash \text{if } (e) \text{ then } c; \text{while } (e) \text{ do } c \text{ else skip} \checkmark} \text{TIF}$$

- Fall BLOCK1_{SS}^T :
 Fall-Annahmen: $\langle c, \sigma[x \mapsto v] \rangle \rightarrow_1 \langle c', \sigma' \rangle$, $\Gamma \vdash \{ \text{var } x = e; c \} \checkmark$, $\sigma :: \Gamma$ und $\mathcal{E} \llbracket e \rrbracket \sigma = v$.
 Induktionsannahme: Für beliebige Γ gilt: Wenn $\Gamma \vdash c \checkmark$ und $\sigma[x \mapsto v] :: \Gamma$, dann $\Gamma \vdash c' \checkmark$.
 Zu zeigen: $\Gamma \vdash \{ \text{var } x = \mathcal{V}^{-1} \llbracket \sigma'(x) \rrbracket; c' \} \checkmark$.

Aus $\Gamma \vdash \{ \text{var } x = e; c \} \checkmark$ erhält man durch Regelinversion (TBLOCK) ein τ mit $\Gamma \vdash e :: \tau$ und $\Gamma[x \mapsto \tau] \vdash c \checkmark$. Mit $\mathcal{E} \llbracket e \rrbracket \sigma = v$ und $\sigma :: \Gamma$ ist wieder $\text{type}(v) = \tau$ nach Lem. 22, also auch $\sigma[x \mapsto v] :: \Gamma[x \mapsto \tau]$, und damit folgt aus der Induktionsannahme mit $\Gamma[x \mapsto \tau]$ für Γ , dass $\Gamma[x \mapsto \tau] \vdash c' \checkmark$.

Mit Lem. 24 gilt $\sigma' :: \Gamma[x \mapsto \tau]$. Damit gilt $\text{type}(\sigma'(x)) = \tau$ und somit $\Gamma \vdash \mathcal{V}^{-1} \llbracket \sigma'(x) \rrbracket :: \tau$ nach Regeln TNUM bzw. TTRUE. Zusammen mit $\Gamma[x \mapsto \tau] \vdash c' \checkmark$ folgt die Behauptung nach Regel TBLOCK. \square

Lem. 23, 24 und 25 zusammen beweisen das Typsicherheitstheorem 21. Folgendes Korollar über die maximalen Ableitungssequenzen ist eine unmittelbare Folgerung daraus:

Korollar 26 (Vollständige Auswertung).

Wenn $\Gamma \vdash c \checkmark$ und $\sigma :: \Gamma$, dann gibt es entweder ein σ' mit $\langle c, \sigma \rangle \xrightarrow{*}_1 \langle \text{skip}, \sigma' \rangle$ oder $\langle c, \sigma \rangle \xrightarrow{\infty}_1$.

Beweis. Angenommen, $\langle c, \sigma \rangle \not\xrightarrow{*}_1$. Dann gibt es ein c' und σ' mit $\langle c, \sigma \rangle \xrightarrow{*}_1 \langle c', \sigma' \rangle$, so dass $\langle c', \sigma' \rangle$ blockiert. Mittels Induktion über die transitive Hülle erhält man aus Lem. 25 und Lem. 24, dass $\Gamma \vdash c' \checkmark$ und $\sigma' :: \Gamma$. Nach dem Fortschrittslemma 23 ist $\langle c', \sigma' \rangle$ aber nur dann blockiert, wenn $c' = \text{skip}$, was zu zeigen war. \square