## 5.5 Getypte Variablen $\mathsf{While}_T$

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

```
x := 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 := true; x := 0`
- `y := true; (if (y) then x := true else x := 0); z := x`
- `x := true; { 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.

### 5.5.1 Typen für $\mathsf{While}_T$

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

**Definition 18 (Ungetypte Ausdrücke).** Die Syntax für (ungetypte) Ausdrücke Exp lautet:

$$\mathsf{Exp}\quad e \quad ::= \quad n \mid x \mid e_1 \texttt{ - } e_2 \mid e_1 \texttt{ * } e_2 \mid \texttt{true} \mid \texttt{false} \mid e_1 \texttt{ <= } e_2 \mid \texttt{not } b \mid b_1 \texttt{ \&\& } b_2$$

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

**Definition 19 (Syntax für Anweisungen in $\mathsf{While}_T$).**

$$\begin{aligned}\mathsf{Com}\quad c \quad ::= \quad & \texttt{skip} \mid x \texttt{ := } e \mid c_1\texttt{; } c_2 \mid \texttt{if (}e\texttt{) then } c_1 \texttt{ else } c_2 \mid \texttt{while (}e\texttt{) do } c \mid \\ & \texttt{\{ var } x \texttt{ = } e\texttt{; } c \texttt{ \}}\end{aligned}$$

**Definition 20 (Typen).** int und bool sind die Typen $\mathbb{T}$ für $\mathsf{While}_T$:

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

Nach Variablenkonvention steht die Metavariable $\tau$ 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 21 (Typkontext).** Ein *Typkontext* $\Gamma :: \mathsf{Var} \Rightarrow \mathbb{T}$ ordnet jeder Variablen einen Typ zu.

Ein Programm besteht also aus der auszuführenden Anweisung $c$ und einem Typkontext $\Gamma$.

**Beispiel 18.** Für die Anweisung

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

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

### 5.5.2 Ein Typsystem für $\mathsf{While}_T$

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

**Definition 22 (Typregeln für Ausdrücke).** Die Typregeln für Ausdrücke Exp sind im Einzelnen:

$$\textsc{TNum}: \Gamma \vdash n :: \mathsf{int} \qquad \textsc{TVar}: \frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau} \qquad \textsc{TTrue}: \Gamma \vdash \texttt{true} :: \mathsf{bool} \qquad \textsc{TFalse}: \Gamma \vdash \texttt{false} :: \mathsf{bool}$$

$$\textsc{TMinus}: \frac{\Gamma \vdash e_1 :: \mathsf{int} \qquad \Gamma \vdash e_2 :: \mathsf{int}}{\Gamma \vdash e_1 \texttt{ - } e_2 :: \mathsf{int}} \qquad \textsc{TTimes}: \frac{\Gamma \vdash e_1 :: \mathsf{int} \qquad \Gamma \vdash e_2 :: \mathsf{int}}{\Gamma \vdash e_1 \texttt{ * } e_2 :: \mathsf{int}}$$

$$\textsc{TLeq}: \frac{\Gamma \vdash e_1 :: \mathsf{int} \quad \Gamma \vdash e_2 :: \mathsf{int}}{\Gamma \vdash e_1 \texttt{ <= } e_2 :: \mathsf{bool}} \qquad \textsc{TNot}: \frac{\Gamma \vdash e :: \mathsf{bool}}{\Gamma \vdash \texttt{not } e :: \mathsf{bool}} \qquad \textsc{TAnd}: \frac{\Gamma \vdash e_1 :: \mathsf{bool} \quad \Gamma \vdash e_2 :: \mathsf{bool}}{\Gamma \vdash e_1 \texttt{ \&\& } e_2 :: \mathsf{bool}}$$

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

$$\dfrac{\dfrac{\dfrac{\Gamma(\mathtt{x}) = \mathsf{int}}{\Gamma \vdash \mathtt{x} :: \mathsf{int}}\,\textsc{TVar} \qquad \dfrac{}{\Gamma \vdash 10 :: \mathsf{int}}\,\textsc{TNum}}{\Gamma \vdash \mathtt{x} \texttt{ <= } 10 :: \mathsf{bool}}\,\textsc{TLeq} \qquad \dfrac{\dfrac{\Gamma(\mathtt{y}) = \mathsf{bool}}{\Gamma \vdash \mathtt{y} :: \mathsf{bool}}\,\textsc{TVar}}{\Gamma \vdash \texttt{not } \mathtt{y} :: \mathsf{bool}}\,\textsc{TNot}}{\Gamma \vdash \texttt{(x <= 10) \&\& (not y)} :: \mathsf{bool}}\,\textsc{TAnd}$$

**Definition 23 (Typkorrektheit von Ausdrücken).** Ein Ausdruck $e$ heißt *typkorrekt* in einem Typkontext $\Gamma$, falls $e$ einen Typ hat, d.h., falls es ein $\tau$ gibt, sodass $\Gamma \vdash e :: \tau$.

**Definition 24 (Typkorrektheit von Anweisungen).** Anweisungen selbst haben keinen Typ, müssen die Ausdrücke aber korrekt verwenden. Typkorrektheit $\Gamma \vdash c\ \surd$ für eine Anweisung $c$ im Typkontext $\Gamma$ ist auch durch ein Regelsystem definiert:

$$\textsc{TSkip}: \Gamma \vdash \texttt{skip}\ \surd \qquad \textsc{TAss}: \frac{\Gamma(x) = \tau \qquad \Gamma \vdash e :: \tau}{\Gamma \vdash x \texttt{ := } e\ \surd} \qquad \textsc{TSeq}: \frac{\Gamma \vdash c_1\ \surd \qquad \Gamma \vdash c_2\ \surd}{\Gamma \vdash c_1\texttt{; } c_2\ \surd}$$

$$\textsc{TIf}: \frac{\Gamma \vdash e :: \mathsf{bool} \qquad \Gamma \vdash c_1\ \surd \qquad \Gamma \vdash c_2\ \surd}{\Gamma \vdash \texttt{if (}e\texttt{) then } c_1 \texttt{ else } c_2\ \surd} \qquad \textsc{TWhile}: \frac{\Gamma \vdash e :: \mathsf{bool} \qquad \Gamma \vdash c\ \surd}{\Gamma \vdash \texttt{while (}e\texttt{) do } c\ \surd}$$

$$\textsc{TBlock}: \frac{\Gamma \vdash e :: \tau \qquad \Gamma[x \mapsto \tau] \vdash c\ \surd}{\Gamma \vdash \texttt{\{ var } x \texttt{ = } e\texttt{; } c \texttt{ \}}\ \surd}$$

**Definition 25 (typisierbar, typkorrekte Programme).** Eine Anweisung $c$ heißt *typisierbar*, wenn es einen Kontext $\Gamma$ mit $\Gamma \vdash c\ \surd$. Ein Programm $P \equiv (c, \Gamma)$ heißt *typkorrekt*, falls $\Gamma \vdash c\ \surd$.

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. { `int x = 5; c` }) oder der Typ für $x$ dürfte sich nicht ändern.

Bemerkung: Die Typsystem-Regeln für $\_ \vdash \_ :: \_$ und $\_ \vdash \_\ \surd$ 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 5.5 sind typkorrekt – bei geeigneter Typdeklaration der globalen Variablen?

### 5.5.3 Small-Step-Semantik für $\mathsf{While}_T$

Zustände müssen jetzt neben Zahlen auch Wahrheitswerte speichern können:

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

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

$$\mathcal{E}\,[\![\_]\!]\,\_ :: \mathsf{Exp} \Rightarrow \Sigma \rightharpoonup \mathbb{Z} + \mathbb{B}$$

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

**Definition 26 (Auswertungsfunktion für Ausdrücke).** Die Auswertungsfunktion $\mathcal{E}\,[\![e]\!]\,\sigma$ kombiniert lediglich die bisherigen Auswertungsfunktionen $\mathcal{A}\,[\![\_]\!]\,\_$ und $\mathcal{B}\,[\![\_]\!]\,\_$:

$$\begin{array}{rcl} \mathcal{E}\,[\![n]\!]\,\sigma & = & \mathcal{N}\,[\![n]\!] \\ \mathcal{E}\,[\![x]\!]\,\sigma & = & \sigma(x) \\ \mathcal{E}\,[\![e_1 \texttt{ - } e_2]\!]\,\sigma & = & \mathcal{E}\,[\![e_1]\!]\,\sigma - \mathcal{E}\,[\![e_2]\!]\,\sigma \\ \mathcal{E}\,[\![e_1 \texttt{ * } e_2]\!]\,\sigma & = & \mathcal{E}\,[\![e_1]\!]\,\sigma \cdot \mathcal{E}\,[\![e_2]\!]\,\sigma \end{array} \qquad \begin{array}{rcl} \mathcal{E}\,[\![\texttt{true}]\!]\,\sigma & = & \mathbf{tt} \\ \mathcal{E}\,[\![\texttt{false}]\!]\,\sigma & = & \mathbf{ff} \\ \mathcal{E}\,[\![e_1 \texttt{ <= } e_2]\!]\,\sigma & = & \mathcal{E}\,[\![e_1]\!]\,\sigma \leq \mathcal{E}\,[\![e_2]\!]\,\sigma \\ \mathcal{E}\,[\![\texttt{not } e]\!]\,\sigma & = & \neg\mathcal{E}\,[\![e]\!]\,\sigma \\ \mathcal{E}\,[\![e_1 \texttt{ \&\& } e_2]\!]\,\sigma & = & \mathcal{E}\,[\![e_1]\!]\,\sigma \wedge \mathcal{E}\,[\![e_2]\!]\,\sigma \end{array}$$

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.** $\mathcal{E}\,[\![\texttt{false \&\& (1 <= true)}]\!]\,\sigma = \bot$, weil $1 \leq \mathbf{tt} = \bot$ und damit auch $\mathbf{ff} \wedge \bot = \bot$.

**Definition 27 (Small-Step-Semantik).** Die Small-Step-Semantik für $\mathsf{While}_T$ besteht nun aus folgenden Regeln:

$$\textsc{Ass}^{\mathrm{T}}_{\mathrm{SS}}\text{:}\ \frac{\mathcal{E}\,[\![e]\!]\,\sigma = v}{\langle x \texttt{ := } e,\ \sigma\rangle \rightarrow_1 \langle \texttt{skip},\ \sigma[x \mapsto v]\rangle}$$

$$\textsc{Seq1}^{\mathrm{T}}_{\mathrm{SS}}\text{:}\ \frac{\langle c_0,\ \sigma\rangle \rightarrow_1 \langle c_0',\ \sigma'\rangle}{\langle c_0\texttt{;}\ c_1,\ \sigma\rangle \rightarrow_1 \langle c_0'\texttt{;}\ c_1,\ \sigma'\rangle} \qquad \textsc{Seq2}^{\mathrm{T}}_{\mathrm{SS}}\text{:}\ \langle \texttt{skip;}\ c,\ \sigma\rangle \rightarrow_1 \langle c,\ \sigma\rangle$$

$$\textsc{IfTT}^{\mathrm{T}}_{\mathrm{SS}}\text{:}\ \frac{\mathcal{E}\,[\![e]\!]\,\sigma = \mathbf{tt}}{\langle \texttt{if (}e\texttt{) then } c_0 \texttt{ else } c_1,\ \sigma\rangle \rightarrow_1 \langle c_0,\ \sigma\rangle}$$

$$\textsc{IfFF}^{\mathrm{T}}_{\mathrm{SS}}\text{:}\ \frac{\mathcal{E}\,[\![e]\!]\,\sigma = \mathbf{ff}}{\langle \texttt{if (}e\texttt{) then } c_0 \texttt{ else } c_1,\ \sigma\rangle \rightarrow_1 \langle c_1,\ \sigma\rangle}$$

$$\textsc{While}^{\mathrm{T}}_{\mathrm{SS}}\text{:}\ \langle \texttt{while (}e\texttt{) do } c,\ \sigma\rangle \rightarrow_1 \langle \texttt{if (}e\texttt{) then } c\texttt{; while (}e\texttt{) do } c \texttt{ else skip},\ \sigma\rangle$$

$$\textsc{Block1}^{\mathrm{T}}_{SS}\text{:}\ \frac{\mathcal{E}\,[\![e]\!]\,\sigma = v \qquad \langle c,\ \sigma[x \mapsto v]\rangle \rightarrow_1 \langle c',\ \sigma'\rangle}{\langle \texttt{\{ var } x \texttt{ = } e\texttt{;}\ c\ \texttt{\}},\ \sigma\rangle \rightarrow_1 \langle \texttt{\{ var } x \texttt{ = } \mathcal{V}^{-1}\,[\![\sigma'(x)]\!]\texttt{;}\ c'\ \texttt{\}},\ \sigma'[x \mapsto \sigma(x)]\rangle}$$

$$\textsc{Block2}^{\mathrm{T}}_{SS}\text{:}\ \frac{\mathcal{E}\,[\![e]\!]\,\sigma \neq \bot}{\langle \texttt{\{ var } x \texttt{ = } e\texttt{; skip \}},\ \sigma\rangle \rightarrow_1 \langle \texttt{skip},\ \sigma\rangle}$$

Dabei ist $\mathcal{V}^{-1}[\![v]\!]$ die Umkehrung von $\mathcal{E}[\![\_]\!]\_$ für Werte $v \in \mathbb{Z} + \mathbb{B}$, also die Verallgemeierung von $\mathcal{N}^{-1}[\![\_]\!]$ auf den neuen Wertebereich für Variablen. Regel $\textsc{Block2}^{\mathrm{T}}_{SS}$ ist nur anwendbar, wenn der Initialisierungsausdruck $e$ keine Typfehler bei der Auswertung hervorruft.

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

### 5.5.4 Typsicherheit von $\mathsf{While}_T$

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

**Definition 28 (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 $\mathsf{While}_T$ ist nicht vollständig. Das Programm `x := 0; x := true` ist nicht typkorrekt, verursacht aber trotzdem keine Laufzeittypfehler.

Um die Typsicherheit von $\mathsf{While}_T$ formulieren zu können, brauchen wir noch zwei Begriffe: Den Typ eines Wertes und Zustandskonformanz.

**Definition 29 (Typ eines Wertes).** Der *Typ* $\mathrm{type}(v)$ eines Wertes $v \in \mathbb{Z} + \mathbb{B}$ ist:

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

**Definition 30 (Zustandskonformanz).** Ein Zustand $\sigma$ ist zu einem Typkontext $\Gamma$ *konformant*, notiert als $\sigma :: \Gamma$, wenn $\mathrm{type}(\sigma(x)) = \Gamma(x)$ für alle $x \in \mathsf{Var}$.

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

**Theorem 18 (Typsicherheit).** Sei $\Gamma \vdash c\ \surd$ und $\sigma :: \Gamma$.
*Progress* Wenn $c \neq \texttt{skip}$, dann gibt es $c'$ und $\sigma'$ mit $\langle c,\ \sigma\rangle \rightarrow_1 \langle c',\ \sigma'\rangle$.
*Preservation* Wenn $\langle c,\ \sigma\rangle \rightarrow_1 \langle c',\ \sigma'\rangle$, dann $\Gamma \vdash c'\ \surd$ und $\sigma' :: \Gamma$.

Den Beweis teilen wir auf die folgenden Hilfslemmata auf:

**Lemma 19 (Typkorrektheit von $\mathcal{E}[\![\_]\!]\_$).**
Wenn $\Gamma \vdash e :: \tau$ und $\sigma :: \Gamma$, dann ist $\mathcal{E}[\![e]\!]\,\sigma$ definiert und $\mathrm{type}(\mathcal{E}[\![e]\!]\,\sigma) = \tau$.

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

- Fälle TNum, TTrue, TFalse: Trivial.
- Fall TVar: Definiertheit ist trivial. Wegen Zustandskonformanz $\sigma :: \Gamma$ gilt: $\text{type}(\mathcal{E}\,[\![e]\!]\,\sigma) = \text{type}(\sigma(x)) = \Gamma(x) = \tau$.
- Fall TMinus:
  Induktionsannahmen: $\mathcal{E}\,[\![e_1]\!]\,\sigma$ und $\mathcal{E}\,[\![e_2]\!]\,\sigma$ sind definiert mit $\text{type}(\mathcal{E}\,[\![e_1]\!]\,\sigma) = \text{type}(\mathcal{E}\,[\![e_2]\!]\,\sigma) = \mathsf{int}$. Nach Definition von type(_) ist somit $\mathcal{E}\,[\![e_1]\!]\,\sigma \in \mathbb{Z}$ und $\mathcal{E}\,[\![e_1]\!]\,\sigma \in \mathbb{Z}$. Also ist auch $\mathcal{E}\,[\![e_1 \texttt{ - } e_2]\!]\,\sigma$ definiert mit $\mathcal{E}\,[\![e_1 \texttt{ - } e_2]\!]\,\sigma \in \mathbb{Z}$, also $\text{type}(\mathcal{E}\,[\![e_1 \texttt{ - } e_2]\!]\,\sigma) = \mathsf{int}$.
- Fälle TTimes, TLeq, TNot, TAnd: Analog. □

**Lemma 20 (Fortschritt).**
Wenn $\Gamma \vdash c\ \surd$ und $\sigma :: \Gamma$ mit $c \neq \texttt{skip}$, dann gibt es $c'$ und $\sigma'$ mit $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$.

*Beweis.* Induktion über $\Gamma \vdash c\ \surd$ oder $c$ direkt – analog zu Lem. 3 ($\sigma$ beliebig). Für die Definiertheit von $\mathcal{E}\,[\![e]\!]\,\sigma$ in den Regelannahmen verwendet man Lem. 19. Bei Blöcken ist wie bei der Sequenz eine Fallunterscheidung über $c = \texttt{skip}$ nötig, für den induktiven Fall $c \neq \texttt{skip}$ braucht man die Induktionshypothese mit $\sigma[x \mapsto v]$ für $\sigma$. □

**Lemma 21 (Erhalt der Zusandskonformanz).**
Wenn $\Gamma \vdash c\ \surd$, $\sigma :: \Gamma$ und $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$, dann $\sigma' :: \Gamma$.

*Beweis.* Regel-Induktion über $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$ ($\Gamma$ beliebig).

- Fall $\text{Ass}^{\text{T}}_{\text{SS}}$: Zu zeigen: Wenn $\Gamma \vdash x \texttt{ := } e\ \surd$, $\sigma :: \Gamma$ und $\mathcal{E}\,[\![e]\!]\,\sigma = v$, dann $\sigma[x \mapsto v] :: \Gamma$.

  Aus $\Gamma \vdash x \texttt{ := } e\ \surd$ erhält man mit Regelinversion (TAss) $\tau$ mit $\Gamma(x) = \tau$ und $\Gamma \vdash e :: \tau$. Aus $\Gamma \vdash e :: \tau$ und $\sigma :: \Gamma$ folgt nach Lem. 19, dass $\text{type}(v) = \tau$. Zusammen mit $\sigma :: \Gamma$ und $\Gamma(x) = \tau$ folgt die Behauptung $\sigma[x \mapsto v] :: \Gamma$.
- Fall $\text{Seq1}^{\text{T}}_{\text{SS}}$: Fall-Annahmen: $\langle c_1,\ \sigma\rangle \to_1 \langle c_1',\ \sigma'\rangle$, $\Gamma \vdash c_1\texttt{ ; }c_2\ \surd$, $\sigma :: \Gamma$.
  Induktionsannahme: Für alle $\Gamma$ gilt: Wenn $\Gamma \vdash c_1\ \surd$ und $\sigma :: \Gamma$, dann $\sigma' :: \Gamma$.
  Zu zeigen: $\sigma' :: \Gamma$.

  Aus $\Gamma \vdash c_1\texttt{ ; }c_2\ \surd$ erhält man durch Regelinversion (TSeq), dass $\Gamma \vdash c_1\ \surd$ und $\Gamma \vdash c_2\ \surd$. Mit $\sigma :: \Gamma$ folgt die Behauptung aus der Induktionsannahme.
- Fälle $\text{Seq2}^{\text{T}}_{\text{SS}}$, $\text{IfTT}^{\text{T}}_{\text{SS}}$, $\text{IfFF}^{\text{T}}_{\text{SS}}$, $\text{While}^{\text{T}}_{\text{SS}}$, $\text{Block2}^{\text{T}}_{SS}$: Trivial, da $\sigma' = \sigma$.
- Fall $\text{Block1}^{\text{T}}_{SS}$:
  Fall-Annahmen: $\mathcal{E}\,[\![e]\!]\,\sigma = v$, $\langle c,\ \sigma[x \mapsto v]\rangle \to_1 \langle c',\ \sigma'\rangle$, $\Gamma \vdash \texttt{\{ var } x \texttt{ = } e\texttt{; } c \texttt{ \}}\ \surd$ und $\sigma :: \Gamma$.
  Induktionsannahme: Für beliebige $\Gamma$ gilt: Wenn $\Gamma \vdash c\ \surd$ und $\sigma[x \mapsto v] :: \Gamma$, dann $\sigma' :: \Gamma$.
  Zu zeigen: $\sigma'[x \mapsto \sigma(x)] :: \Gamma$.

  Aus $\Gamma \vdash \texttt{\{ var } x \texttt{ = } e\texttt{; } c \texttt{ \}}\ \surd$ erhält man durch Regelinversion (TBlock) $\tau$ mit $\Gamma \vdash e :: \tau$ und $\Gamma[x \mapsto \tau] \vdash c\ \surd$. Aus $\Gamma \vdash e :: \tau$ und $\sigma :: \Gamma$ folgt mit Lem. 19, 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 $\Gamma$ 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 22 (Subject reduction).** Wenn $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$, $\Gamma \vdash c\ \surd$ und $\sigma :: \Gamma$, dann $\Gamma \vdash c'\ \surd$.

*Beweis.* Regel-Induktion über $\langle c,\ \sigma\rangle \to_1 \langle c',\ \sigma'\rangle$ ($\Gamma$ beliebig).

- Fälle $\text{Ass}^{\text{T}}_{\text{SS}}$, $\text{Block2}^{\text{T}}_{SS}$: Trivial mit Regel TSkip.
- Fall $\text{Seq1}^{\text{T}}_{\text{SS}}$:
  Induktionsannahme: Wenn $\Gamma \vdash c_1\ \surd$ und $\sigma :: \Gamma$, dann $\Gamma \vdash c_1'\ \surd$.
  Zu zeigen: Wenn $\Gamma \vdash c_1\texttt{ ; }c_2\ \surd$ und $\sigma :: \Gamma$, dann $\Gamma \vdash c_1'\texttt{ ; }c_2\ \surd$.

Aus $\Gamma \vdash c_1\texttt{;}\ c_2\ \surd$ erhält man durch Regelinversion (TSeq), dass $\Gamma \vdash c_1\ \surd$ und $\Gamma \vdash c_2\ \surd$. Mit $\sigma :: \Gamma$ folgt aus der Induktionsannahme, dass $\Gamma \vdash c_1'\ \surd$. Zusammen mit $\Gamma \vdash c_2\ \surd$ folgt die Behauptung mit Regel TSeq.

- Fälle $\text{Seq2}^{\text{T}}_{\text{SS}}$, $\text{IfTT}^{\text{T}}_{\text{SS}}$, $\text{IfFF}^{\text{T}}_{\text{SS}}$: Trivial mit Regelinversion (TSeq bzw. TIf).
- Fall $\text{While}^{\text{T}}_{\text{SS}}$: Aus $\Gamma \vdash \texttt{while (}e\texttt{) do }c\ \surd$ folgt mit Regelinversion, dass $\Gamma \vdash e :: \mathsf{bool}$ und $\Gamma \vdash c\ \surd$. Damit gilt:

$$\frac{\Gamma \vdash e :: \mathsf{bool} \qquad \dfrac{\Gamma \vdash c\ \surd \qquad \Gamma \vdash \texttt{while (}e\texttt{) do }c\ \surd}{\Gamma \vdash c\texttt{; while (}e\texttt{) do }c\ \surd}\ \text{TSeq} \qquad \dfrac{}{\Gamma \vdash \texttt{skip}\ \surd}\ \text{TSkip}}{\Gamma \vdash \texttt{if (}e\texttt{) then }c\texttt{; while (}e\texttt{) do }c\texttt{ else skip}\ \surd}\ \text{TIf}$$

- Fall $\text{Block1}^{\text{T}}_{SS}$:
  Fall-Annahmen: $\langle c,\ \sigma[x \mapsto v]\rangle \rightarrow_1 \langle c',\ \sigma'\rangle$, $\Gamma \vdash \texttt{\{ var }x\texttt{ = }e\texttt{;}\ c\texttt{ \}}\ \surd$, $\sigma :: \Gamma$ und $\mathcal{E}\,[\![e]\!]\,\sigma = v$.
  Induktionsannahme: Für beliebige $\Gamma$ gilt: Wenn $\Gamma \vdash c\ \surd$ und $\sigma[x \mapsto v] :: \Gamma$, dann $\Gamma \vdash c'\ \surd$.
  Zu zeigen: $\Gamma \vdash \texttt{\{ var }x\texttt{ = }\mathcal{V}^{-1}\,[\![\sigma'(x)]\!]\texttt{;}\ c'\texttt{ \}}\ \surd$.

  Aus $\Gamma \vdash \texttt{\{ var }x\texttt{ = }e\texttt{;}\ c\texttt{ \}}\ \surd$ erhält man durch Regelinversion (TBlock) ein $\tau$ mit $\Gamma \vdash e :: \tau$ und $\Gamma[x \mapsto \tau] \vdash c\ \surd$. Mit $\mathcal{E}\,[\![e]\!]\,\sigma = v$ und $\sigma :: \Gamma$ ist wieder $\text{type}(v) = \tau$ nach Lem. 19, also auch $\sigma[x \mapsto v] :: \Gamma[x \mapsto \tau]$, und damit folgt aus der Induktionsannahme mit $\Gamma[x \mapsto \tau]$ für $\Gamma$, dass $\Gamma[x \mapsto \tau] \vdash c'\ \surd$.

  Mit Lem. 21 gilt $\sigma' :: \Gamma[x \mapsto \tau]$. Damit gilt $\text{type}(\sigma'(x)) = \tau$ und somit $\Gamma \vdash \mathcal{V}^{-1}\,[\![\sigma'(x)]\!] :: \tau$ nach Regeln TNum, TTrue bzw. TFalse. Zusammen mit $\Gamma[x \mapsto \tau] \vdash c'\ \surd$ folgt die Behauptung nach Regel TBlock. □

Lem. 20, 21 und 22 zusammen beweisen das Typsicherheitstheorem 18. Folgendes Korrolar über die maximalen Ableitungssequenzen ist eine unmittelbare Folgerung daraus:

**Korollar 23 (Vollständige Auswertung).**
Wenn $\Gamma \vdash c\ \surd$ und $\sigma :: \Gamma$, dann gibt es entweder ein $\sigma'$ mit $\langle c,\ \sigma\rangle \xrightarrow{*}_1 \langle \texttt{skip},\ \sigma'\rangle$ oder $\langle c,\ \sigma\rangle \xrightarrow{\infty}_1$.

*Beweis.* Angenommen, $\langle c,\ \sigma\rangle \not\xrightarrow{\infty}_1$. Dann gibt es ein $c'$ und $\sigma'$ 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. 22 und Lem. 21, dass $\Gamma \vdash c'\ \surd$ und $\sigma' :: \Gamma$. Nach dem Fortschrittslemma 20 ist $\langle c',\ \sigma'\rangle$ aber nur dann blockiert, wenn $c' = \texttt{skip}$, was zu zeigen war. □