1 Syntax der Logik

1.1 Typen

Vorgegeben sei die Menge *Type* aller *Typen* τ der Programmiersprache. Dann definieren wir

- die Menge *SType* aller *statischen Typen* θ
- die Menge *LType* aller *logischen Typen* π

durch

$$\begin{array}{rcl} \theta & ::= & \tau \\ & | & \theta_1 \xrightarrow{t} \theta_2 \\ & | & \theta_1 \ \textbf{sequence} \\ & | & \theta_1 \times \ldots \times \theta_n \quad (n \geq 2) \end{array}$$

$$\begin{array}{rcl} \pi & ::= & \theta \\ & | & \tau \ \textbf{expr} \\ & | & \theta \ \textbf{assn} \end{array}$$

und als syntaktischen Zucker verwenden wir

$$\textbf{assn} \quad \text{für} \quad \textbf{unit assn}$$

1.2 Terme und Formeln

Vorgegeben sei die Menge *Exp* aller *Ausdrücke* e und die Menge *Val* aller *Werte* v der Programmiersprache. Dann definieren wir

- die Menge F aller *Funktionszeichen* f
- die Menge *Term* aller *Terme* t
- die Menge *Assn* aller *assertions* p, q, r, s und
- die Menge *Form* aller *(Hoare-)Formeln* h

durch

$$
\begin{array}{rcll}
f & ::= & + \mid - \mid * \mid < \mid > \mid \leq \mid \geq & \\
 & \mid & \mathit{cons} \mid \mathit{elem} \mid \mathit{length} \mid \ldots & \\
 & & & \\
t & ::= & v & \\
 & \mid & f & \\
 & \mid & t_1\, t_2 & \\
 & \mid & \lambda id : \theta.\, t_1 & \\
 & \mid & (t_1, \ldots, t_n) & (n \geq 2) \\
 & & & \\
p, q, r, s & ::= & t_1 \mapsto t_2 & \\
 & \mid & \mathit{disj}(p, t) & \\
 & \mid & p\, t & \\
 & \mid & \lambda id : \theta.\, p & \\
 & \mid & \neg\, p_1 & \\
 & \mid & p_1 \wedge p_2 & \\
 & \mid & \exists\, id : \theta.\, p_1 & \\
 & \mid & h & \\
 & & & \\
h & ::= & \{p\}\, e\, \{q\} & \\
 & \mid & t_1 = t_2 & \\
 & \mid & \mathit{view}\, p & \\
 & \mid & \neg\, h_1 & \\
 & \mid & h_1 \wedge h_2 & \\
 & \mid & \exists\, id : \theta.\, h_1 &
\end{array}
$$

und als syntaktischen Zucker verwenden wir

$$
\begin{array}{rcl}
p_1 \vee p_2 & \text{für} & \neg\, (\neg\, p_1 \wedge \neg\, p_2) \\
p_1 \Rightarrow p_2 & \text{für} & \neg\, p_1 \vee p_2 \\
\forall\, id : \theta.\, p & \text{für} & \neg\, \exists\, id : \theta.\, \neg\, p \\
h_1 \vee h_2 & \text{für} & \neg\, (\neg\, h_1 \wedge \neg\, h_2) \\
h_1 \Rightarrow h_2 & \text{für} & \neg\, h_1 \vee h_2 \\
\forall\, id : \theta.\, h & \text{für} & \neg\, \exists\, id : \theta.\, \neg\, h \\
\{p\}\, e\, \{\textbf{returns}\ id : \tau.\, q\} & \text{für} & \{p\}\, e\, \{\lambda id : \tau.\, q\} \\
\{p\}\, e\, \{q\} & \text{für} & \{p\}\, e\, \{\lambda id : \textbf{unit}.\, q\} \\
\{p\} & \text{für} & \{\mathit{true}\}\, ()\, \{p\}
\end{array}
$$

2 Typregeln für die Logik

Eine *Typumgebung* für die Logik ist eine endliche partielle Funktion

$$\Gamma : \mathit{Id} \hookrightarrow \mathit{SType}$$

Typurteile für die Logik sind von der Form

$$\begin{array}{rl}
\Gamma \rhd e :: \tau \ \mathbf{expr} & \text{für Ausdrücke } e \\
\Gamma \rhd t :: \theta & \text{für Terme } t \\
\Gamma \rhd p :: \mathbf{assn} \text{ oder } \Gamma \rhd p :: \theta \ \mathbf{assn} & \text{für assertions } p \\
\Gamma \rhd h & \text{für Formeln } h
\end{array}$$

Ein Typurteil $\Gamma \rhd e :: \tau \ \mathbf{expr}$ für Ausdrücke betrachten wir als *gültig*, wenn das entsprechende Typurteil $\Gamma \rhd e :: \tau$ in der Programmiersprache herleitbar ist. Für die übrigen Typurteile geben wir neue Regeln an.

Die gültigen Typurteile für assertions erhalten wir mit den Regeln

$$\text{(APP)} \quad \frac{\Gamma \rhd p :: \theta \ \mathbf{assn} \quad \Gamma \rhd t :: \theta}{\Gamma \rhd p\, t :: \mathbf{assn}}$$

$$\text{(CONT)} \quad \frac{\Gamma \rhd t_1 :: \tau \, \mathbf{ref} \quad \Gamma \rhd t_2 :: \tau}{\Gamma \rhd t_1 \mapsto t_2 :: \mathbf{assn}}$$

$$\text{(ABSTR)} \quad \frac{\Gamma[\theta/id] \rhd p :: \mathbf{assn}}{\Gamma \rhd \lambda id : \theta.\, p :: \theta \ \mathbf{assn}}$$

$$\text{(DISJ)} \quad \frac{\Gamma \rhd p :: \theta_1 \ \mathbf{assn} \quad \Gamma \rhd t :: \theta_2}{\Gamma \rhd \mathit{disj}(p, t) :: \mathbf{assn}}$$

$$\text{(NOT)} \quad \frac{\Gamma \rhd p :: \mathbf{assn}}{\Gamma \rhd \neg p :: \mathbf{assn}}$$

$$\text{(AND)} \quad \frac{\Gamma \rhd p_1 :: \mathbf{assn} \quad \Gamma \rhd p_2 :: \mathbf{assn}}{\Gamma \rhd p_1 \wedge p_2 :: \mathbf{assn}}$$

$$\text{(EXISTS)} \quad \frac{\Gamma[\theta/id] \rhd p :: \mathbf{assn}}{\Gamma \rhd \exists\, id : \theta.\, p :: \mathbf{assn}}$$

$$\text{(HOARE)} \quad \frac{\Gamma \rhd h}{\Gamma \rhd h :: \mathbf{assn}}$$

und die für Hoare-Formeln mit

(TC) $$\frac{\Gamma \rhd p :: \textbf{assn} \quad \Gamma \rhd e :: \tau\ \textbf{expr} \quad \Gamma \rhd q :: \tau\ \textbf{assn}}{\Gamma \rhd \{p\}\, e\, \{q\}}$$

(EQ) $$\frac{\Gamma \rhd t_1 :: \theta \quad \Gamma \rhd t_2 :: \theta}{\Gamma \rhd t_1 = t_2}$$

(VIEW) $$\frac{\Gamma \rhd p :: \theta\ \textbf{assn}}{\Gamma \rhd \mathit{view}\, p :: \textbf{assn}}$$

(H-NOT) $$\frac{\Gamma \rhd h}{\Gamma \rhd \neg h}$$

(H-AND) $$\frac{\Gamma \rhd h_1 \quad \Gamma \rhd h_2}{\Gamma \rhd h_1 \wedge h_2}$$

(H-EXISTS) $$\frac{\Gamma[\theta/id] \rhd h}{\Gamma \rhd \exists\, id : \theta.\, h}$$

3 Speicherzustände und Erreichbarkeit

Für jeden Typ τ sei eine unendliche Menge Loc^τ vorgegeben, deren Elemente $X, Y, \ldots$ wir als *Speicherplätze* vom Typ τ bezeichnen. Wir setzen voraus, dass die Mengen Loc^τ paarweise disjunkt sind und definieren

$$Loc = \bigcup_{\tau \in \mathit{Type}} Loc^\tau$$

Wir schreiben $locns(e)$ für die Menge aller im Ausdruck e vorkommenden Speicherplätze und Val^τ für die Menge aller abgeschlossenen Werte vom Typ τ (in denen Speicherplätze vorkommen dürfen). Unter einem *Speicherzustand* verstehen wir eine endliche partielle Funktion

$$\sigma : Loc \to \bigcup_{\tau \in \mathit{Type}} [\![\tau]\!]$$

mit den Eigenschaften

- $\sigma(Loc^\tau) \subseteq Val^\tau$ für jeden Typ τ
- $locns(\sigma(X)) \subseteq dom(\sigma)$ für alle $X \in Loc$

Es wird also vorausgesetzt, dass ein Zustand stets *wohlgetypt* ist und *keine dangling references* enthält.

Mit *Store* bezeichnen wir die Menge aller Zustände und für jede endliche Menge $L \subseteq Loc$ definieren wir

$$Store(L) = \{\sigma \in Store \mid L \subseteq dom(\sigma)\}$$

Für $\sigma \in Store(L)$ seien die Mengen $reach_i(L, \sigma) \subseteq Loc$ für alle $i \in \mathbb{N}$ und die Menge $reach(L, \sigma) \subseteq Loc$ definiert durch

$$\begin{array}{rcl} reach_0(L,\sigma) & = & L \\ reach_{i+1}(L,\sigma) & = & reach_i(L,\sigma) \ \cup \ \bigcup_{X \in reach_i(L,\sigma)} locns(\sigma(X)) \\ reach(L,\sigma) & = & \bigcup_{i \in \mathbb{N}} reach_i(L,\sigma) \end{array}$$

$reach(L, \sigma)$ enthält alle Speicherplätze, die man von der Menge L aus im Zustand σ *erreichen* kann. Da ein Zustand $\sigma \in Store(L)$ keine dangling references enthält, gilt stets

$$reach(L, \sigma) \subseteq dom(\sigma)$$

Definition 1 (L-Äquivalenz) Zwei Zustände $\sigma, \sigma' \in Store(L)$ heißen *L-äquivalent* (Schreibweise: $\sigma \equiv_L \sigma'$), wenn folgendes gilt:

(a) $reach(L, \sigma) = reach(L, \sigma')$

(b) σ und σ' stimmen auf $reach(L, \sigma)$ überein

4 Semantik der Logik

Jedem $\pi \in LType$ wird ein semantischer Bereich $[\![\pi]\!]$ zugeordnet, insbesondere

$$\begin{array}{rcl} [\![\tau]\!] & = & Val^{\tau} \\ [\![\mathbf{assn}]\!] & = & Store \hookrightarrow Bool \\ [\![\tau\ \mathbf{assn}]\!] & = & Val^{\tau} \times Store \hookrightarrow Bool \\ [\![\tau\ \mathbf{expr}]\!] & = & Store \hookrightarrow Val^{\tau} \times Store \end{array}$$

Eine Funktion $f \in [\![\tau\ \mathbf{expr}]\!]$ heißt *total korrekt* bezüglich $\varphi \in [\![\mathbf{assn}]\!]$ und $\psi \in [\![\tau\ \mathbf{assn}]\!]$, wenn für alle $\sigma \in dom(\varphi)$ gilt:

$$\text{Wenn } \varphi(\sigma) = true, \text{ dann ist } \sigma \in dom(f) \text{ und } \psi(f\sigma) = true.$$

Eine *Umgebung* ist eine endliche partielle Funktion $\rho : \mathit{Id} \hookrightarrow \bigcup_{\pi \in \mathit{LType}} [\![\pi]\!]$. ρ *passt* zur Typumgebung Γ (Schreibweise: $\Gamma \models \rho$), wenn gilt

- $\mathit{dom}(\rho) = \mathit{dom}(\Gamma)$
- $\rho(\mathit{id}) \in [\![\Gamma(\mathit{id})]\!]$ für alle $\mathit{id} \in \mathit{dom}(\rho)$

Mit $\mathit{Env}(\Gamma)$ bezeichnen wir die Menge aller zu Γ passenden Umgebungen.

Ein Zustand σ *passt* zur Umgebung ρ (Schreibweise: $\rho \models \sigma$), wenn gilt

- $\mathit{locns}(\rho(\mathit{id})) \subseteq \mathit{dom}(\sigma)$ für alle $\mathit{id} \in \mathit{dom}(\rho)$
- $\mathit{locns}(\sigma(X)) \subseteq \mathit{dom}(\sigma)$ für alle $X \in \mathit{dom}(\sigma)$

Mit $\mathit{Store}(\rho)$ bezeichnen wir die Menge aller zu ρ passenden Speicherzustände.

Für die (noch zu definierende) Semantik wohlgetypter preconditions setzen wir voraus, dass stets

$$\mathit{dom}([\![\Gamma \triangleright p :: \mathbf{assn}]\!]\,\rho) = \mathit{Store}(\rho)$$

gilt. Die Semantik wohlgetypter postconditions wird dann definiert durch

$$\begin{aligned}
&[\![\Gamma \triangleright \mathbf{returns}\ \mathit{id} : \tau.\, p :: \tau\ \mathbf{assn}]\!]\,\rho = \psi, \text{ wobei} \\
&\mathit{dom}(\psi) = \{(v,\sigma) \in \mathit{Val}^{\tau} \times \mathit{Store} \mid \sigma \in \mathit{Store}(\rho[v/\mathit{id}])\} \\
&\psi(v,\sigma) = [\![\Gamma[\tau/\mathit{id}] \triangleright p :: \mathbf{assn}]\!]\,\rho[v/\mathit{id}]\,\sigma \quad \text{für alle } (v,\sigma) \in \mathit{dom}(\psi)
\end{aligned}$$

Die Semantik wohlgetypter Ausdrücke wird auf die big step Semantik zurückgeführt durch

$$[\![\Gamma \triangleright e :: \tau]\!]\,\rho\,\sigma = \begin{cases} (v,\sigma') & \text{falls } (e\rho,\sigma) \Downarrow (v,\sigma') \\ \text{undefiniert} & \text{falls } (e\rho,\sigma) \not\Downarrow \end{cases}$$

wobei $e\rho$ den (abgeschlossenen) Ausdruck $e[\rho(\mathit{id})/\mathit{id}]_{\mathit{id} \in \mathit{free}\,(e)}$ bezeichnet.

Darauf aufbauend wird die Semantik von Hoare-Tripeln definiert durch

$$\begin{aligned}
&[\![\Gamma \triangleright \{p\}\, e\, \{\mathbf{returns}\ \mathit{id} : \tau.\, q\} :: \mathbf{prop}]\!]\,\rho = \mathit{true} \iff \\
&\quad [\![\Gamma \triangleright e :: \tau\ \mathbf{expr}]\!]\,\rho \text{ ist total korrekt bezüglich} \\
&\quad [\![\Gamma \triangleright p :: \mathbf{assn}]\!]\,\rho \text{ und } [\![\Gamma \triangleright \mathbf{returns}\ \mathit{id} : \tau.\, q :: \tau\ \mathbf{assn}]\!]\,\rho
\end{aligned}$$

und die Semantik aller übrigen Formeln wird im Sinne der Prädikatenlogik definiert, z.B.

$$
\begin{aligned}
&[\![\Gamma \triangleright \forall\, id : \tau.\, h :: \mathbf{prop}]\!]\, \rho = true \;\;\Leftrightarrow \\
&\quad [\![\Gamma[\tau/id] \triangleright h :: \mathbf{prop}]\!]\, \rho[v/id] = true \;\; \text{für alle } v \in [\![\tau]\!]
\end{aligned}
$$

5 Der Kalkül

5.1 Regeln für die Programmiersprache

Diese Regeln beschreiben die Semantik der einzelnen Ausdrücke unserer Programmiersprache.

(VAL) $\forall\, x : \tau.\, \{\mathit{true}\}\, x\, \{\textbf{returns}\ y : \tau.\, y = x\}$

(REF-1) $\forall\, x : \tau.\, \{\mathit{true}\}\, \mathit{ref}\ x\, \{\textbf{returns}\ y : \tau\, \textbf{ref}.\, y \mapsto x\}$

(REF-2) $\forall\, x : \tau. \forall\, z : \tau'.\, \{\mathit{true}\}\, \mathit{ref}\ x\, \{\textbf{returns}\ y : \tau\, \textbf{ref}.\, \mathit{disj}(y, z)\}$

(REF-3) $\forall\, x : \tau. \forall\, w : \theta\ \textbf{assn}.\, \{\mathit{true}\}\, \mathit{ref}\ x\, \{\textbf{returns}\ y : \tau\, \textbf{ref}.\, \mathit{disj}(w, y)\}$

(REF-4) $\forall\, x : \tau.\, \{p\}\, \mathit{ref}\ x\, \{p\}$

(DEREF-1) $\forall\, x : \tau\, \textbf{ref}. \forall\, y : \tau.\, \{x \mapsto y\}\ !\, x\, \{\textbf{returns}\ z : \tau.\, z = y\}$

(DEREF-2) $\forall\, x : \tau\, \textbf{ref}.\, \{p\}\ !\, x\, \{p\}$

(ASSIGN-1) $\forall\, x : \tau\, \textbf{ref}. \forall\, y : \tau.\, \{\mathit{true}\}\, x := y\, \{x \mapsto y\}$

(ASSIGN-2) $\forall\, x : \tau\, \textbf{ref}. \forall\, y : \tau.\, \mathit{view}\, p \wedge \mathit{disj}(p, x) \Rightarrow \{p\}\, x := y\, \{p\}$

(BETA-V) $\forall\, x : \tau.\ \{p\}\, e\, \{\textbf{returns}\ y : \tau'.\, q\}$
$\qquad\qquad \Rightarrow \{p\}\, (\lambda x : \tau.\, e)\, x\, \{\textbf{returns}\ y : \tau'.\, q\}$

(APP)
$$\begin{aligned}
&\{p\}\, e_1\, \{\textbf{returns}\ x : \tau \to \tau'.\, q\} \ \wedge \\
&\ (\forall\, x : \tau \to \tau'.\, \{q\}\, e_2\, \{\textbf{returns}\ y : \tau.\, r\}) \ \wedge \\
&\ (\forall\, x : \tau \to \tau',\ y : \tau.\, \{r\}\, x\, y\, \{\textbf{returns}\ z : \tau'.\, s\}) \\
&\Rightarrow\ \{p\}\, e_1\, e_2\, \{\textbf{returns}\ z : \tau'.\, s\}
\end{aligned}$$

falls $x \notin \mathit{free}\,(e_2),\ x, y \notin \mathit{free}\,(s)$

(IF-TRUE) $\{p\}\, e_0\, \{\textbf{returns}\ x : \textbf{bool}.\, q \wedge x = \mathit{true}\} \wedge$
$\{q\}\, e_1\, \{\textbf{returns}\ y : \tau.\, r\}$
$\Rightarrow \{p\}\, \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2\, \{\textbf{returns}\ y : \tau.\, r\}$

falls $x \notin \mathit{free}\,(q)$

(IF-FALSE) $\{p\}\, e_0\, \{\textbf{returns}\ x : \textbf{bool}.\, q \wedge x = \mathit{false}\} \wedge$
$\{q\}\, e_1\, \{\textbf{returns}\ y : \tau.\, r\}$
$\Rightarrow \{p\}\, \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2\, \{\textbf{returns}\ y : \tau.\, r\}$

falls $x \notin \mathit{free}\,(q)$

5.2 Regeln für totale Korrektheit

Diese Regeln erlauben es—unabhängig vom speziellen Ausdruck—über totale Korrektheit zu argumentieren.

(CONSEQ) $(\{p \Rightarrow q\} \;\wedge\; \{q\}\, e\, \{\textbf{returns}\ x : \tau.\, r\} \;\wedge\; \{\forall\, x : \tau.\, r \Rightarrow s\})$
$\Rightarrow\; \{p\}\, e\, \{\textbf{returns}\ x : \tau.\, s\}$

(PRE-H) $(h \Rightarrow \{p\}\, e\, \{\textbf{returns}\ x : \tau.\, q\})$
$\Rightarrow\; \{h \wedge p\}\, e\, \{\textbf{returns}\ x : \tau.\, q\}$

(PRE-$\vee$) $(\{p\}\, e\, \{\textbf{returns}\ x : \tau.\, r\} \;\wedge\; \{q\}\, e\, \{\textbf{returns}\ x : \tau.\, r\})$
$\Rightarrow\; \{p \vee q\}\, e\, \{\textbf{returns}\ x : \tau.\, r\}$

(PRE-$\exists$) $(\forall\, x : \tau.\, \{p\}\, e\, \{\textbf{returns}\ y : \tau'.\, q\})$
$\Rightarrow\; \{\exists\, x : \tau.\, p\}\, e\, \{\textbf{returns}\ y : \tau'.\, q\}$

falls $x \notin \mathit{free}\,(e) \cup \mathit{free}\,(q)$

(POST-$\wedge$) $(\{p\}\, e\, \{\textbf{returns}\ x : \tau.\, q\} \;\wedge\; \{p\}\, e\, \{\textbf{returns}\ x : \tau.\, r\})$
$\Rightarrow\; \{p\}\, e\, \{\textbf{returns}\ x : \tau.\, q \wedge r\}$

(POST-$\forall$) $(\forall\, x : \tau.\, \{p\}\, e\, \{\textbf{returns}\ y : \tau'.\, q\})$
$\Rightarrow\; \{p\}\, e\, \{\textbf{returns}\ y : \tau'.\, \forall\, y : \tau'.\, q\}$

falls $x \notin \mathit{free}\,(p) \cup \mathit{free}\,(e)$

6 Abgeleitete Regeln

Aus (VAL) folgt sofort

$$\text{(VAL}'\text{)} \quad \{\mathit{true}\}\, v\, \{\mathbf{returns}\; y : \tau.\, y = v\}$$

7 Spezifikation von Objekten

In der Spezifikation eines Objekts beschreiben wir, wie das Objekt auf Nachrichten reagieren soll, d.h. wie die Methodenaufrufe den abstrakten Zustand des Objekts verändern sollen. Da der abstrakte Zustand durch eine *view* beschrieben wird, besitzt die Spezifikation neben dem Objekt auch eine *view* als Parameter. Eine Spezifikation für Zählerobjekte könnte dann z.B. so aussehen:

$$\begin{aligned} \mathit{counter\_spec} \quad = \quad & \lambda c : \langle \mathit{inc} : \mathbf{unit}; \mathit{get} : \mathbf{int}\rangle. \\ & \quad \lambda w : \mathbf{int\ assn}. \\ & \qquad (\forall\, i : \mathbf{int}.\, \{w\, i\}\, c\#\mathit{inc}\, \{w\, (i+1)\}) \wedge \\ & \qquad (\forall\, i : \mathbf{int}.\, \{w\, i\}\, c\#\mathit{get}\, \{\mathbf{returns}\; j : \mathbf{int}.\, w\, i \wedge j = i\}) \end{aligned}$$

Eine so formulierte Objektspezifikation können wir dann bei der Spezifikation eines Objekt*generators* verwenden, z.B. lässt sich ein Zählergenerator *new_counter* so spezifizieren:

$$\begin{aligned} \{\mathit{true}\}\, \mathit{new\_counter}\, ()\, \{\mathbf{returns}\; c : \langle \mathit{inc} : \mathbf{unit}; \mathit{get} : \mathbf{int}\rangle. \\ \exists\, w : \mathbf{int\ assn}.\, w\, 0 \wedge\ \mathit{counter\_spec}\, c\, w\} \end{aligned}$$

Darüber hinaus verlangen wir von einem Zählergenerator noch, dass er bei jedem Aufruf einen *neuen* Zähler liefert, d.h. einen Zähler, der disjunkt zu allen bisher bekannten *views* ist, und dessen eigene *view* disjunkt zu allen bisher bekannten Elementen der Programmiersprache ist. Die erste Eigenschaft lässt sich beschreiben durch

$$\begin{aligned} & \forall w : \theta\ \mathbf{assn}. \\ & \quad \{\mathit{true}\}\, \mathit{new\_counter}\, ()\, \{\mathbf{returns}\; c : \langle \mathit{inc} : \mathbf{unit}; \mathit{get} : \mathbf{int}\rangle.\, \mathit{disj}(w, c)\} \end{aligned}$$

Die naheliegende Implementierung für einen Zählergenerator ist

$$\begin{array}{rcl} \mathit{new\_counter\_1} & = & \lambda().\,\textbf{object} \\ & & \quad \textbf{val}\ x = \mathit{ref}\ 0 \\ & & \quad \textbf{method}\ \mathit{inc} = x\ := \ !\,x + 1 \\ & & \quad \textbf{method}\ \mathit{get} = !\,x \\ & & \textbf{end} \end{array}$$

Für diese Implementierung gilt folgende *konkrete Spezifikation*

$$\begin{array}{l} \{\mathit{true}\}\ \mathit{new\_counter\_1}\,()\,\{\textbf{returns}\ c : \langle \mathit{inc} : \textbf{unit}; \mathit{get} : \textbf{int}\rangle. \\ \qquad \exists\, x : \textbf{int ref}. \\ \qquad\quad !\,x = 0\ \wedge \\ \qquad\quad \forall\, i : \textbf{int}.\,\{\,!\,x = i\}\ c\#\mathit{inc}\,\{\,!\,x = i + 1\}\ \wedge \\ \qquad\quad \forall\, i : \textbf{int}.\,\{\,!\,x = i\}\ c\#\mathit{get}\,\{\textbf{returns}\ i\}\} \end{array}$$

Um daraus die gewünschte *abstrakte Spezifikation*

$$\{\mathit{true}\}\ \mathit{new\_counter\_1}\,()\,\{\textbf{returns}\ c : \langle \mathit{inc} : \textbf{unit}; \mathit{get} : \textbf{int}\rangle.\ \mathit{counter\_spec}\ c\}$$

zu erhalten, verwenden wir die *consequence rule* zur Abschwächung der *post-condition*. Also bleibt folgende *verification condition* zu beweisen

$$\begin{array}{l} \forall\, c : \langle \mathit{inc} : \textbf{unit}; \mathit{get} : \textbf{int}\rangle. \\ \quad (\exists\, x : \textbf{int ref}. \ldots x \ldots) \Rightarrow (\exists\, \mathit{cview} : \textbf{int} \to \textbf{bool}. \ldots \mathit{cview} \ldots) \end{array}$$

Sie lässt sich mit prädikatenlogischen Regeln zurückführen auf

$$\ldots x \ldots \Rightarrow (\exists\, \mathit{cview} : \textbf{int} \to \textbf{bool}. \ldots \mathit{cview} \ldots)$$

und letzteres beweist man, indem man $\mathit{cview} = \lambda i : \textbf{int}.\ !\,x = i$ als Beispiel für die Existenzaussage wählt.
Eine alternative Implementierung für einen Zählergenerator ist

$$\begin{array}{rcl} \mathit{new\_counter\_2} & = & \lambda().\,\textbf{object} \\ & & \quad \textbf{val}\ x = \mathit{ref}\ 0 \\ & & \quad \textbf{method}\ \mathit{inc} = x\ := \ !\,x + 2 \\ & & \quad \textbf{method}\ \mathit{get} = !\,x \bmod 2 \\ & & \textbf{end} \end{array}$$

Für *new_counter_2* gilt eine andere *konkrete Spezifikation*, nämlich

$$\{true\}\ new\_counter\_2\,()\ \{\mathbf{returns}\ c : \langle inc : \mathbf{unit}; get : \mathbf{int}\rangle.$$
$$\exists\, x : \mathbf{int\,ref}.$$
$$!\,x = 0\ \wedge$$
$$\forall\, i : \mathbf{int}.\ \{\,!\,x = 2 * i\}\ c\#inc\ \{\,!\,x = 2 * (i+1)\}\ \wedge$$
$$\forall\, i : \mathbf{int}.\ \{\,!\,x = 2 * i\}\ c\#get\ \{\mathbf{returns}\ i\}\}$$

Aber nach wie vor erhält man die gleiche *abstrakte Spezifikation*

$$\{true\}\ new\_counter\_2\,()\ \{\mathbf{returns}\ c : \langle inc : \mathbf{unit}; get : \mathbf{int}\rangle.\ counter\_spec\ c\}$$

indem man die *postcondition* mit Hilfe der *consequence rule* abschwächt. Die verbleibende *verification condition*

$$\forall\, c : \langle inc : \mathbf{unit}; get : \mathbf{int}\rangle.$$
$$(\exists\, x : \mathbf{int\,ref}.\ldots x \ldots) \Rightarrow (\exists\, cview : \mathbf{int} \to \mathbf{bool}.\ldots cview \ldots)$$

beweist man dieses Mal mit $cview = \lambda i : \mathbf{int}.\ !\,x = 2 * i$ als Beispiel für die Existenzaussage.

Basierend auf der Zählerspezifikation lässt sich nun die Korrektheit von *abstrakten Programmen* beweisen, die mit einem Zähler arbeiten, z.B.

$$\forall\, c : \langle inc : \mathbf{unit}; get : \mathbf{int}\rangle.$$
$$counter\_spec\ c \Rightarrow \{true\}\ c\#inc;\ c\#inc;\ c\#get\ \{\mathbf{returns}\ 2\}$$

Daraus erhält man dann die Korrektheit von *konkreten Programmen*, die mit einer speziellen Implementierung eines Zählers arbeiten, z.B.

$$\{true\}\ \mathbf{let}\ c = new\_counter\_1\,()\ \mathbf{in}\ c\#inc;\ c\#inc;\ c\#get\ \{\mathbf{returns}\ 2\}$$

oder

$$\{true\}\ \mathbf{let}\ c = new\_counter\_2\,()\ \mathbf{in}\ c\#inc;\ c\#inc;\ c\#get\ \{\mathbf{returns}\ 2\}$$

7.0.1 Bemerkung:

Mit den bisherigen Überlegungen lässt sich nur die Korrektheit von abstrakten Programmen beweisen, die auf einem einzigen Objekt arbeiten. Sobald mehr als ein Objekt im Spiel ist, muss man wissen, dass jedes Objekt ausschließlich über seine eigenen Methoden "erreichbar" ist. Dazu bedarf es einer weiteren Spezifikation, die sich (vermutlich) unabhängig von der bisherigen formulieren lässt.

8 Fallstudie: stack-Objekte

Wir wollen stack-Objekte mit Hilfe von verketteten Listen implementieren. Ein rekursiver Typ für verkettete Listen lässt sich deklarieren durch

$$\begin{array}{rcl} \textbf{type}\ \mathit{cell} & = & \mathit{Nil} \\ & | & \mathit{Cons}\ \textbf{of int} * \mathit{cell}\ \textbf{ref} \end{array}$$

und ein entsprechender *stack*-Generator

$$\mathit{new\_stack} : \textbf{unit} \to \langle \mathit{push} : \textbf{int} \to \textbf{unit};\ \mathit{pop} : \textbf{unit};\ \mathit{top} : \textbf{int} \rangle$$

durch

$$\begin{array}{l} \textbf{let}\ \mathit{new\_stack} = \lambda().\ \textbf{object} \\ \qquad\qquad\qquad\qquad \textbf{val}\ r = \mathit{Nil} \\ \qquad\qquad\qquad\qquad \textbf{method}\ \mathit{push} = \lambda x : \textbf{int}.\ r\ := \ \mathit{Cons}(x, \mathit{ref}\,(\,!\,r)) \\ \qquad\qquad\qquad\qquad \textbf{method}\ \mathit{pop} = \textbf{let}\ \mathit{Cons}(x, r') = !\,r\ \textbf{in}\ r\ :=\ !\,r' \\ \qquad\qquad\qquad\qquad \textbf{method}\ \mathit{top} = \textbf{let}\ \mathit{Cons}(x, r') = !\,r\ \textbf{in}\ x \\ \qquad\qquad\qquad\qquad \textbf{end} \end{array}$$

Um nachzuweisen, dass *new_stack* die übliche *abstrakte* Spezifikation erfüllt, muss eine passende *view* auf den generierten Objekten definiert werden. Dazu definiert man zunächst ein Prädikat

$$\mathit{contains} : \mathit{cell} \xrightarrow{t} \textbf{int sequence assn}$$

induktiv durch

$$\begin{array}{rcl} \mathit{contains}\,r\,[\,] & = & (r \mapsto \mathit{Nil}) \\ \mathit{contains}\,r\,(\mathit{cons}(x, l)) & = & \exists\, r' : \mathit{cell}\,\textbf{ref}\,[r].\, r \mapsto \mathit{Cons}(x, r') \ \wedge\ \mathit{contains}\,r'\,l \end{array}$$

und beweist damit die folgende *konkrete* Spezifikation:

$$\begin{array}{l} \{\mathit{true}\}\ \mathit{new\_stack}\,()\ \{\textbf{returns}\ \mathit{st} : \langle \mathit{push} : \textbf{int} \to \textbf{unit};\ \mathit{pop} : \textbf{unit};\ \mathit{top} : \textbf{int} \rangle. \\ \qquad\qquad\qquad\qquad \exists\, r : \mathit{cell}\,\textbf{ref}\,[\mathit{st}]. \\ \qquad\qquad\qquad\qquad\quad \mathit{contains}\,r\,[\,] \\ \qquad\qquad\qquad\qquad\quad \wedge\ \forall\, x : \textbf{int}, l : \textbf{int sequence}. \\ \qquad\qquad\qquad\qquad\qquad\quad \{\mathit{contains}\,r\,l\}\ \mathit{st}\#\mathit{push}\,x\ \{\mathit{contains}\,r\,(\mathit{cons}(x, l))\} \\ \qquad\qquad\qquad\qquad\quad \wedge\ \forall\, x : \textbf{int}, l : \textbf{int sequence}. \\ \qquad\qquad\qquad\qquad\qquad\quad \{\mathit{contains}\,r\,(\mathit{cons}(x, l))\}\ \mathit{st}\#\mathit{pop}\ \{\mathit{contains}\,r\,l\} \\ \qquad\qquad\qquad\qquad\quad \wedge\ \forall\, x : \textbf{int}, l : \textbf{int sequence}. \\ \qquad\qquad\qquad\qquad\qquad\quad \{\mathit{contains}\,r\,(\mathit{cons}(x, l))\}\ \mathit{st}\#\mathit{top}\ \{\textbf{returns}\ y : \textbf{int}. \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad y = x\ \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{contains}\,r\,(\mathit{cons}(x, l))\}\} \end{array}$$

Mit semantischen Mitteln lässt sich zeigen, dass $[\![\mathit{contains}\, r]\!]\rho$ stets $\{\rho(r)\}$-definierbar, also von $\{\rho(r)\}$ erreichbar ist. Insbesondere ist $\mathit{contains}\, r$ in obiger Spezifikation vom Objekt st erreichbar, weil das Existenz-quantifizierte r von st erreichbar ist. Also kann man eine *view* w : **int sequence assn** definieren durch

$$w \quad = \quad \mathit{contains}\, r$$

und erhält so aus der *konkreten* die gewünschte *abstrakte* Spezifikation

$$
\begin{array}{l}
\{\mathit{true}\}\; \mathit{new\_stack}\,()\; \{\mathbf{returns}\; st : \langle \mathit{push} : \mathbf{int} \to \mathbf{unit};\; \mathit{pop} : \mathbf{unit};\; \mathit{top} : \mathbf{int}\rangle. \\
\qquad\qquad\qquad\qquad \exists\, w : \mathbf{int\ sequence\ assn}\; [st]. \\
\qquad\qquad\qquad\qquad\quad w\,[\,] \\
\qquad\qquad\qquad\qquad\quad \wedge\, \forall\, x : \mathbf{int}, l : \mathbf{int\ sequence}. \\
\qquad\qquad\qquad\qquad\qquad \{w\, l\}\; st\#\mathit{push}\, x\; \{w\,(\mathit{cons}(x, l))\} \\
\qquad\qquad\qquad\qquad\quad \wedge\, \forall\, x : \mathbf{int}, l : \mathbf{int\ sequence}. \\
\qquad\qquad\qquad\qquad\qquad \{w\,(\mathit{cons}(x, l))\}\; st\#\mathit{pop}\; \{w\, r\, l\} \\
\qquad\qquad\qquad\qquad\quad \wedge\, \forall\, x : \mathbf{int}, l : \mathbf{int\ sequence}. \\
\qquad\qquad\qquad\qquad\qquad \{w\,(\mathit{cons}(x, l))\}\; st\#\mathit{top}\; \{\mathbf{returns}\; y : \mathbf{int}. \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y = x\, \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad w\,(\mathit{cons}(x, l))\}
\end{array}
$$