Bedeutung und Beweis von C-Programmen
Lehrveranstaltung, Uni Bremen, SoSe 08

Christoph Lüth 1. August 2008

1 Präliminarien

1.1 Partielle Abbildungen

Gegeben zwei Mengen X und Y ist eine *partielle Abbildung* $f : X \rightharpoonup Y$ eine Funktion von X nach Y, die für jedes $x \in X$ entweder genau ein $y \in Y$ oder ein ausgezeichnetes Element $\perp$ zurückgibt; wir schreiben dafür fx. Mengentheoretisch können wir $X \rightharpoonup Y$ als eine linkseindeutige Relation $X \times Y$ charakterisieren; hier favorisieren wir eine axiomatische Definition.

Die *leere Abbildung* $\emptyset : X \rightharpoonup Y$ ist definiert als $\emptyset(x) = \perp$ für alle $x \in X$.

Die *punktweise Definition* von f an der Stelle $x \in X$ mit dem Wert $y \in Y$ ist definiert als

$$f(x \mapsto y)x' = \begin{cases} y & x = x' \\ fx' & x \neq x' \end{cases} \tag{1}$$

Für partielle Abbildungen $f, g : X \rightharpoonup Y$ gilt die *extensionale Gleichheit*, d.h. f und g sind gleich, wenn sie für alle Werte in x dasselbe Ergebnis liefern:

$$f = g \iff \forall x.fx = gx \tag{2}$$

Partielle Abbildungen sind durch folgende vier Basiseigenschaften charakterisiert (wobei $x_1 \neq x_2$):

$$f(x \mapsto y)x = y \tag{3}$$
$$f(x \mapsto y_1)(x \mapsto y_2) = f(x \mapsto y_2) \tag{4}$$
$$f(x_1 \mapsto y)x_2 = fx_2 \tag{5}$$
$$f(x_1 \mapsto y_1)(x_2 \mapsto y_2) = f(x_2 \mapsto y_2)(x_1 \mapsto y_1) \tag{6}$$
$$\tag{7}$$

Diese Eigenschaften folgen direkt aus der Definition (1), oder lassen sich leicht durch Extensionalität (2) beweisen.

Kontexte

Ein Kontext ist eine partielle Abbildung von Bezeichner auf bestimmte Werte. Konkret haben wir *Typkontexte*, welche Bezeichner auf Typen abbilden, *Variablenkontexte*, welche Bezeichner auf Adressen (*Loc*, Abschnitt 3.1) abbilden, und *Funktionskontexte*, welche Funktionen auf Zustandsübergänge abbilden (Abschnitt 3.2.4).

Für Kontext verwenden wir folgende Notation: $\emptyset$ ist der leere Kontext, Γf bezeichnet das Lesen, und $\Gamma, (x : t)$ fügt die Abbildung des Bezeichners x auf Typ, Adresse oder Zustandsübergang t hinzu.

2 Ein Typsystem für C

Die Typbestimmung erfolgt in Form einer *Typableitung* (Judgement) $\Gamma \vdash e : t$, wobei Γ ein sog. Kontext ist, e ein Ausdruck und t ein Typ. Der Kontext hält dabei den Typ von anderswo im

Programm definierten Bezeichnern fest; das sind beispielsweise globale Variabeln, oder für die Bestimmung des Typs eines Ausdrucks innerhalb einer Funktion, die lokalen Variablen.

Typableitung wird durch einen Satz induktiver Regeln definiert; eine Typableitung ist genau dann gültig, wenn sie sich mit diesen Regeln herleiten läßt. (Eine nicht herleitbare Ableitung ist also auch nicht gültig.)

2.1 Abstrakte Syntax für Typen

Die abstrakte Syntax ist im wesentlichen der soch durch die Parsierung des Programmes nach der konkreten Syntaxbaum. Aufgrund der etwas skurrilen Syntax für Deklarationen und Typen in C (s. Standard), die Ausdrücke wie

```
short long float char double int short short short
```

grammatikalisch zuläßt, benötigen wir für Typen eine etwas abstraktere Syntax. Diese sind in §6.2.5 definiert.

*Vereinfachung:*Folgende Typen werden im folgenden nicht betrachtet: komplexe Typen (`complex`),erweiterte Ganzzahltypen (*extended integer types*), Vereinigungstypen (*union*).

Wir unterscheiden zwischen den Basistypen (*basic types*) und den abgeleiteten Typen (*derived types*).

2.1.1 Basic Types

$$
\begin{aligned}
T_{Int} &= \{\texttt{signed char}, \texttt{short int}, \texttt{int}, \texttt{long int}, \texttt{long long int}\} \\
T_{UnsignedInt} &= \{\texttt{unsigned char}, \texttt{signed short int}, \ldots\} \\
T_{Basic} &= T_{Int} \cup T_{UnsignedInt} \\
T_{Int} &= \{\texttt{char}\} \cup T_{SignedInt} \cup T_{UnsignedInt} \\
T_{Floating} &= \{\texttt{float}, \texttt{double}, \texttt{long double}\} \\
T_{Basic} &= T_{Int} \cup T_{Floating}
\end{aligned}
$$

Das Schlüsselwort `signed` ist nur für `char` relevant, ansonsten ist beispielsweise `signed int` dasselbe wie `int`. Die Funktion *signed* bildet den nichtvorzeichenbehafteten Ganzzahltypen auf den entsprechenden vorzeichenbehafteten Ganzzahltypen ab; die Funktion *unsigned* ist ihr Inverses:

$$
\begin{aligned}
\mathit{signed} &: T_{UnsignedInt} \rightarrow T_{SignedInt} \\
\mathit{signed}(\texttt{unsigned char}) &= \texttt{signed char} \\
\mathit{signed}(\texttt{unsigned short int}) &= \texttt{short int} \\
\mathit{signed}(\texttt{unsigned int}) &= \texttt{int} \\
\mathit{signed}(\texttt{unsigned long int}) &= \texttt{long int} \\
\mathit{signed}(\texttt{unsigned long long int}) &= \texttt{long long int} \\
\mathit{unsigned} &: T_{SignedInt} \rightarrow T_{UnsignedInt} \\
\mathit{unsigned}(\texttt{signed char}) &= \texttt{unsigned char} \\
\mathit{unsigned}(\texttt{short int}) &= \texttt{unsigned short int} \\
\mathit{unsigned}(\texttt{int}) &= \texttt{unsigned int} \\
\mathit{unsigned}(\texttt{long int}) &= \texttt{unsigned long int} \\
\mathit{unsigned}(\texttt{long long int}) &= \texttt{unsigned long long int}
\end{aligned}
$$

2.1.2 Derived Types

Abgeleitete Typen werden durch (möglicherweise rekursive) Anwendung von Regeln konstruiert. Damit ergibt sich die Menge aller Typen T wie folgt:

$$\begin{aligned} t \in T_{Basic} &\longrightarrow t \in T \\ t \in T &\longrightarrow ptr\ t \in T \\ t \in T, n \in \mathbb{N} &\longrightarrow array\ t\ n \in T \\ t_1, \ldots, t_n \in T, i_1, \ldots, i_n \in Idt &\longrightarrow \langle\!|\ i_1 : t_1, \ldots, i_n : t_n\ |\!\rangle \in T \\ &\texttt{void} \in T \\ T_{Scalar} &= T_{Arith} \cup \{ptr\ t \mid t \in T\} \end{aligned}$$

2.1.3 Größe eines Typen

Die Funktion *size* ist ein für unsere Zwecke vereinfachte Funktion zur Berechnung der Größe eines Typen. Wir nehmen vereinfachend an, dass alle Basiswerte gleich groß sind; natürlich würden sich in einer konkreten Implementierung die Größen unterscheiden (nach den im Standard angegeben Einschränkungen), aber für unsere Zwecke reicht diese Definition:

$$\begin{aligned} size &: T \rightharpoonup \mathbb{N} \\ size(t) &= 1 \qquad (t \in T_{Basic}) \\ size(ptr\ t) &= 1 \\ size(\langle\!|\ i_1 : t_1 \ldots, i_n : t_n\ |\!\rangle) &= \sum_{j=1}^{n} size(t_j) \\ size(array\ t\ n) &= size(t) \cdot n \end{aligned}$$

Die Größe von `void` ist nicht definiert.

2.2 Typkonversionen

Die Sprache C kennt zwei Typkonversionen, die implizit angewendet werden (und die als einer der wesentlichen Fehlerquellen der Sprache gelten). Wir geben hier eine formale Definition.

2.2.1 Ganzzahl-Umwandlungsrang

Der *integer conversion rank* nach §6.3.1.1 (1) definiert, ob eine ganze Zahl in eine andere konvertiert werden kann. Formal definieren wir ihn als die Relation $\prec$ auf T_{Int}, welche die kleinste Relation ist, die folgende induktive Regeln erfüllt:

$$\begin{aligned} \texttt{signed char} &\prec \texttt{short int} \\ \texttt{short int} &\prec \texttt{int} \\ \texttt{int} &\prec \texttt{long int} \\ \texttt{long int} &\prec \texttt{long long int} \\ \texttt{char} &\prec \texttt{short int} \\ \texttt{char} &\prec \texttt{unsigned short int} \\ s \in T_{SignedInt}, signed(s) \prec t &\longrightarrow s \prec t \qquad (8) \\ t \in T_{SignedInt}, s \prec signed(t) &\longrightarrow s \prec t \qquad (9) \\ r \prec s, s \prec t &\longrightarrow r \prec t \qquad (10) \end{aligned}$$

Regel (10) besagt, dass $\prec$ transitiv ist, und Regeln (8) (9) besagen, dass vorzeichenbehaftete und nichtvorzeichenbehaftete Ganzzahlen den gleichen Umwandlungsrang besitzen.

2.2.2 Integer Promotion

Hierunter ist nicht eine ganze Zahl mit einem Doktorgrad zu verstehen, sondern die Konversion von ganzen Zahlen auf eine einheitliche Mindestgröße nach §6.3.1.1 (2). Wir definieren diese als Relation $\lhd_i$ auf T_{Int}:

$$\begin{aligned} s \prec \texttt{int}, s \in T_{SignedInt} &\longrightarrow s \lhd_i \texttt{int} \\ s \prec \texttt{unsigned int}, s \in T_{UnsignedInt} &\longrightarrow s \lhd_i \texttt{unsigned int} \end{aligned}$$

Das bedeutet nichts weiter, als dass alle Typen mit kleinerem Umwandlungsrang als (`signed`) `int` vorzeichenerhaltend in diesen konvertiert werden. **Fehlt: Alle größeren Integers werden auf sich selbst konvertiert, das ist für $\lhd_a$ nötig.**

2.2.3 Arithmetische Konversion

Die anspruchsvollste der automatischen Typkonversionen definiert sich nach §6.3.1.8 (*usual arithmetic conversions*) als eine ternäre (dreistellige) Relation $\lhd_a$ auf den arithmetischen Typen T_{Arith} wie folgt:

$$(t, \texttt{long double}) \lhd_a \texttt{long double} \tag{11}$$
$$(\texttt{long double}, t) \lhd_a \texttt{long double} \tag{12}$$
$$t \neq \texttt{long double} \longrightarrow (t, \texttt{double}) \lhd_a \texttt{double} \tag{13}$$
$$t \neq \texttt{long double} \longrightarrow (\texttt{double}, t) \lhd_a \texttt{double} \tag{14}$$
$$t \neq \texttt{double}, \texttt{long double} \longrightarrow (t, \texttt{float}) \lhd_a \texttt{float} \tag{15}$$
$$t \neq \texttt{double}, \texttt{long double} \longrightarrow (\texttt{float}, t) \lhd_a \texttt{float} \tag{16}$$
$$t_1 \lhd_i s_1, t_s \lhd_i s_2 \longrightarrow (t_1, t_2) \lhd_a \phi(t_1, t_2) \tag{17}$$

In (17) impliziert die Vorbedingung, dass $s_1, s_2 \in T_{Int}$. Die Hilfsfunktion $Conv$ ist wie folgt definiert:

$$\phi(s_1, s_2) = \begin{cases} s_1 & s_1 = s_2 \\ s_1 & s_1 \in T_{Int}, s_2 \in T_{Int}, s_2 \prec s_1 \\ s_2 & s_1 \in T_{Int}, s_2 \in T_{Int}, s_1 \prec s_2 \\ s_1 & s_1 \in T_{UnsignedInt}, s_2 \in T_{UnsignedInt}, s_2 \prec s_1 \\ s_2 & s_1 \in T_{UnsignedInt}, s_2 \in T_{UnsignedInt}, s_1 \prec s_2 \\ s_1 & s_1 \in T_{UnsignedInt}, s_2 \in T_{SignedInt}, s_2 \prec s_1 \\ s_2 & s_1 \in T_{SignedInt}, s_2 \in T_{UnsignedInt}, s_1 \prec s_2 \\ s_2 & s_1 \in T_{UnsignedInt}, s_2 \in T_{SignedInt}, val(s_1) \subseteq val(s_2) \\ s_1 & s_1 \in T_{SignedInt}, s_2 \in T_{UnsignedInt}, val(s_2) \subseteq val(s_1) \\ unsigned(s_2) & s_1 \in T_{UnsignedInt}, s_2 \in T_{SignedInt} \\ unsigned(s_1) & s_1 \in T_{SignedInt}, s_2 \in T_{UnsignedInt} \end{cases} \tag{18}$$

Informell lassen sich die arithmetischen Konversionen wie folgt charakterisieren [1]: alles wird in die größte Fließkommazahl — wenn nötig — oder ganze Zahl konvertiert, vorzeichenbehaftet falls das ohne Wertverlust möglich ist.

2.3 Regeln

Die folgenden Regeln dienen zur Herleitung der Typen. Sie formalisieren die in §6.5 angegebene informelle Herleitung des Typen.

Primäre Ausdrücke §6.5.1

$$\frac{x \in Idt \qquad \Gamma(x) = t}{\Gamma \vdash x : t} \quad \text{(Identifier)}$$

$$\frac{l \in Lit \qquad l \text{ hat Typ } t}{\Gamma \vdash l : t} \quad \text{(Literale)}$$

$$\frac{}{\Gamma \vdash \texttt{!}s\texttt{!} : ptr\ \texttt{char}} \quad \text{(Zeichenketten)}$$

$$\frac{}{\Gamma \vdash \texttt{0NULL} : ptr\ t} \quad \text{(Nullzeiger)}$$

Postfix-Operatoren §6.5.2

$$\frac{\Gamma \vdash e : ptr\ t \qquad \Gamma \vdash i : s \qquad s \in T_{Int}}{\Gamma \vdash e\texttt{[}i\texttt{]} : t} \quad \text{(Feldzugriff)}$$

$$\frac{\Gamma \vdash f : s_1 \times \ldots s_n t \qquad \Gamma \vdash e_i : s_i}{\Gamma \vdash f\texttt{(}e_1, \ldots, e_n\texttt{)} : t} \quad \text{(Funktionsaufruf)}$$

$$\frac{\Gamma \vdash e : \langle\!|\ i_1 : t_1, \ldots, i_n : t_n\ |\!\rangle \qquad j \in 1 \ldots n}{\Gamma \vdash e\texttt{.}i_j : t_i} \quad \text{(Selektor)}$$

$$\frac{\Gamma \vdash e : ptr\ \langle\!|\ i_1 : t_1, \ldots, i_n : t_n\ |\!\rangle \qquad j \in 1 \ldots n}{\Gamma \vdash e\texttt{->}i_j : t_i} \quad \text{(Zeigerselektor)}$$

Unäre Operatoren §6.5.3

$$\frac{\Gamma \vdash e : t \qquad t \in T_{Scalar}}{\Gamma \vdash e\texttt{++} : t} \quad \text{(Postinkrement)}$$

$$\frac{\Gamma \vdash e : t \qquad t \in T_{Scalar}}{\Gamma \vdash e\texttt{--} : t} \quad \text{(Postdekrement)}$$

$$\frac{\Gamma \vdash e : t \qquad t \in T_{Scalar}}{\Gamma \vdash \texttt{++}e : t} \quad \text{(Präinkrement)}$$

$$\frac{\Gamma \vdash e : t \qquad t \in T_{Scalar}}{\Gamma \vdash \texttt{--}e : t} \quad \text{(Prädekrement)}$$

$$\frac{\Gamma \vdash e : ptr\ t}{\Gamma \vdash *e : t} \quad \text{(Derefenzierung)}$$

$$\frac{\Gamma \vdash e : ptr\ t}{\Gamma \vdash *e : t} \quad \text{(Dereferenzierung)}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \&e : ptr\ t} \quad \text{(Adresse)}$$

$$\frac{\Gamma \vdash e : t \qquad t \in T_{Int} \qquad \lhd_i\ ts}{\Gamma \vdash \ \ e : s} \quad \text{(Komplement)}$$

$$\frac{\Gamma \vdash e : t \qquad t \in T_{Arith} \qquad \lhd_i\ ts}{\Gamma \vdash \texttt{-}e : s} \quad \text{(Unäres Minus)}$$

- Hier fehlt das unäre Plus (`+e`), was aber irgendwie nutzlos ist.

$$\frac{\Gamma \vdash e : t \qquad t \in T_{Scalar} \qquad \lhd_i\ ts}{\Gamma \vdash \texttt{!}e : s} \quad \text{(Negation)}$$

$$\frac{}{\Gamma \vdash \texttt{sizeof}(t) : \texttt{size\_t}} \quad \text{(Grösse)}$$

Der Typ `size_t` ist in `<stddef.h>` definiert.

Typkonversion §6.5.4

$$\frac{\Gamma \vdash e : s \qquad s, t \in T_{Scalar}}{\Gamma \vdash (t)e : t} \quad \text{(Typkonversion)}$$

Multiplikative Operatoren §6.5.5

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash e * f : t} \quad \text{(Multiplikation)}$$

Die Bedingung $\lhd_a\ s_1, s_2 t$ impliziert, dass $s_1, s_2, t \in T_{Arith}$.

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash e / f : t} \quad \text{(Division)}$$

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad s_1, s_2 \in T_{Int} \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash e \% f : t} \quad \text{(Modulus)}$$

Additive Operatoren §6.5.6

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash e\texttt{+}f : t} \qquad \text{(Addition-Arith)}$$

$$\frac{\Gamma \vdash e : ptr\ t \qquad \Gamma \vdash f : s \qquad s \in T_{Int}}{\Gamma \vdash e\texttt{+}f : ptr\ t} \qquad \text{(Addition-Zeiger-R)}$$

$$\frac{\Gamma \vdash e : s \qquad \Gamma \vdash f : ptr\ t \qquad s \in T_{Int}}{\Gamma \vdash e\texttt{+}f : ptr\ t} \qquad \text{(Addition-Zeiger-L)}$$

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash e\texttt{-}f : t} \qquad \text{(Subtraktion-Arith)}$$

$$\frac{\Gamma \vdash e : ptr\ t \qquad \Gamma \vdash f : ptr\ t}{\Gamma \vdash e\texttt{-}f : \texttt{ptrdiff\_t}} \qquad \text{(Subtraktion-Zeiger)}$$

Bitweises Verschieben §6.5.7

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad \lhd_i\ s_1 t_1 \qquad s_2 \in T_{Int}}{\Gamma \vdash e\texttt{<<}f : t_1} \qquad \text{(Verschieben-L)}$$

Relationen §6.5.8

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash e\texttt{<}f : \texttt{int}} \qquad \text{(Kleiner-Arith)}$$

$$\frac{\Gamma \vdash e : ptr\ t \qquad \Gamma \vdash f : ptr\ t}{\Gamma \vdash e\texttt{<}f : \texttt{int}} \qquad \text{(Kleiner-Zeiger)}$$

- Sechs weitere, analoge Regeln für `<=`, `>`, `>=` (je 2).

Gleichheit §6.5.9

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash e\texttt{==}f : \texttt{int}} \qquad \text{(Gleich-Arith)}$$

$$\frac{\Gamma \vdash e : ptr\ t \qquad \Gamma \vdash f : ptr\ t}{\Gamma \vdash e\texttt{==}f : \texttt{int}} \qquad \text{(Gleich-Zeiger-1)}$$

$$\frac{\Gamma \vdash e : ptr\ t \qquad \Gamma \vdash f : ptr\ \texttt{void}}{\Gamma \vdash e\texttt{==}f : \texttt{int}} \qquad \text{(Gleich-Zeiger-2a)}$$

$$\frac{\Gamma \vdash e : ptr\ \texttt{void} \qquad \Gamma \vdash f : ptr\ t}{\Gamma \vdash e\texttt{==}f : \texttt{int}} \qquad \text{(Gleich-Zeiger-2b)}$$

- Vier weitere, analoge Regeln für `!=`.

Binäre Logische Operatoren §6.5.10– 12

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad s_1, s_2 \in T_{Int} \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash e\ \texttt{\&}\ f : t}$$

(Binäre Konjunktion)

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad s_1, s_2 \in T_{Int} \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash e\ \texttt{|}\ f : t}$$

(Binäre Disjunktion)

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad s_1, s_2 \in T_{Int} \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash e\ \texttt{\^{}}\ f : t}$$

(Binäre Ausschliessende Disjunktion)

Logische Operatoren §6.5.13– 14

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad s_1, s_2 \in T_{Scalar}}{\Gamma \vdash e\ \texttt{\&\&}\ f : \texttt{int}}$$

(Konjunktion)

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad s_1, s_2 \in T_{Scalar}}{\Gamma \vdash e\ \texttt{||}\ f : \texttt{int}}$$

(Disjunktion)

Fallunterscheidung §6.5.15

$$\frac{\Gamma \vdash c : s \qquad \Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2 \qquad s \in T_{Scalar} \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash c\ \texttt{?}\ e\ \texttt{:}\ f : t}$$

(Fallunterscheidung-1)

$$\frac{\Gamma \vdash c : s \qquad \Gamma \vdash e : t \qquad \Gamma \vdash f : t \qquad s \in T_{Scalar} \qquad t = \langle |\ i_1 : t_1, \ldots, i_n : t_n\ | \rangle}{\Gamma \vdash c\ \texttt{?}\ e\ \texttt{:}\ f : t}$$

(Fallunterscheidung-2)

$$\frac{\Gamma \vdash c : s \qquad \Gamma \vdash e : ptr\ t \qquad \Gamma \vdash f : ptr\ t \qquad s \in T_{Scalar}}{\Gamma \vdash c\ \texttt{?}\ e\ \texttt{:}\ f : ptr\ t}$$

(Fallunterscheidung-3a)

$$\frac{\Gamma \vdash c : s \qquad \Gamma \vdash e : ptr\ t \qquad \Gamma \vdash f : ptr\ \texttt{void} \qquad s \in T_{Scalar}}{\Gamma \vdash c\ \texttt{?}\ e\ \texttt{:}\ f : ptr\ t}$$

(Fallunterscheidung-3b)

$$\frac{\Gamma \vdash c : s \qquad \Gamma \vdash e : ptr\ \texttt{void} \qquad \Gamma \vdash f : ptr\ t \qquad s \in T_{Scalar}}{\Gamma \vdash c\ \texttt{?}\ e\ \texttt{:}\ f : ptr\ t}$$

(Fallunterscheidung-3c)

$$\frac{\Gamma \vdash c : s \qquad \Gamma \vdash e : \texttt{void} \qquad \Gamma \vdash f : \texttt{void} \qquad s \in T_{Scalar} \qquad \lhd_a\ s_1, s_2 t}{\Gamma \vdash c\ \texttt{?}\ e\ \texttt{:}\ f : \texttt{void}}$$

(Fallunterscheidung-4)

Zuweisung §6.5.16

$$\frac{\Gamma \vdash e : s_1 \qquad \Gamma \vdash f : s_2}{\Gamma \vdash e\ \texttt{=}\ f : s_1} \qquad \text{(Zuweisung)}$$

wobei s_1, s_2 mindestens einer der folgenden Bedingungen erfüllt:

$$s_1, s_2 \in T_{Arith} \tag{19}$$
$$s_1 = \langle |\ i_1 : t_1, \ldots, i_n : t_n\ | \rangle, s_2 = \langle |\ i_1 : t_1, \ldots, i_n : t_n\ | \rangle \tag{20}$$
$$s_1 = ptr\ t, s_2 = ptr\ t \tag{21}$$
$$s_1 = ptr\ \texttt{void}, s_2 = ptr\ t \tag{22}$$
$$s_1 = ptr\ t, s_2 = ptr\ \texttt{void} \tag{23}$$

$$\frac{\Gamma \vdash e\ \texttt{=}\ e\texttt{*}\ f : s}{\Gamma \vdash e\ \texttt{*=}\ f : s} \qquad \text{(Zuweisungsoperatoren)}$$

- Die Regel für Zuweisungsoperatoren ist zulässig, weil der einzige Unterschied zwischen e `*=` f und e `=` e`*` f ist, dass bei ersten e nur einmal ausgewertet wird (§6.5.16.2 (3)).
- Es fehlen die Regeln für die anderen Zuweisungsoperatoren, also `/=`, `%=`, `+=`, `-=`, `<<=`, `&=`, `^=`, `|=`.
- Der Komma-Operator (§6.5.17) wird nicht weiter betrachtet; die Typableitungsregel ist trivial.

3 Eine Denotationale Semantik für C

Eine denotationale Semantik bildet jedes Programm P auf einen Zustandsübergang $\Sigma \to \Sigma$ ab, wobei Σ der Systemzustand (im wesentlichen der Speicher) ist.Für eine denotationale Semantik müssen wir also erst eine mathematische Modellierung des Speichermodells aus dem Standard angeben.

Die mathematischen Grundlagen der denotationalen Semantik einschließlich Erklärung und Motivation finden sich beispielsweise in [2]; hier geben wir eine für C spezifische Semantik.

3.1 Speichermodellierung

Der Standard beschreibt einen strukturierten Speicher, in dem Basisobjekte (`char`, `short int`, `int`, `long int`, `long long int`, `float`, `double`, `long double`), strukturierte Objekte (Felder, Strukturen) und Zeiger gespeichert werden.

Unser Speichermodell bietet keinen flachen Adressraum (in dem der gesamte Speicher hintereinander liegt, adressiert von 0 bis n), sondern einen zweistufiges Adress-Schema, in dem lokale und globale Variablen die erste Stufe der Adressierung bilden, und Indizierung eine zweite. (Das ist mit dem Standard konform §6.2.6.1.) Interessant ist besonders die rekursive Abhängigkeit zwischen Werten (Val) und Adressen (Loc); das ist für die Sprache C charakteristisch, in der Zeiger

Werte wie alle anderen (*first class citizens*) sind.

$$\begin{aligned} \mathit{BaseLoc} &= \{\mathit{Global}\} \times \mathit{Idt} \cup \{\mathit{Local}\} \times \mathbb{N} \\ \mathit{Loc} &= \mathit{BaseLoc} \times \mathbb{N} \\ \mathit{Val} &= \mathbb{Z} \cup \mathbb{R} \cup \mathit{Loc}_* \end{aligned}$$

Hierbei bedeutet die Notation $X_* = X \uplus \{*\}$, d.h. wir fügen der Menge X (hier *Loc*) ein ausgezeichnetes Element hinzu (welches hier den Zeiger `NULL` modelliert). Eine Adresse $l \in \mathit{Loc}$ besteht also aus einer *Basisadresse* (l, i), wobei entweder $l = \mathit{Global}$, dann ist i ein Bezeichner, oder $l = \mathit{Local}$, dann ist i ein natürliche Zahol, sowie einem *Offset* n. Für ein Feld ist n der Index in die Einträge (siehe unten). *Vereinfachung:*: Wir modellieren alle Ganzzahltypen als ganze Zahlen ($\mathbb{Z}$) und Fließkommazahlen durch reelle Zahlen ($\mathbb{R}$). Das ist eine Überapproxmation, erlaubt uns aber kurz und knapp auf existierende mathematische Konzepte zurückzugreifen, ohne erst die (nicht-triviale) Modellierung von Fließkommaarithmetik angeben zu müssen. (Der Standard schreibt hier nichts vor, allerdings sind beliebig große Zahlen unhandlich zu implementieren.) Wir rechnen außerdem ohne Füllelemente und *alignment* in Strukturen; der Standard erlaubt, dass in Strukturen Füllelemente (*padding*) eingefügt werden können, um Adressen auf bestimmte Wortgrenzen (*alignment*) ausrichten zu können. Solange man allerdings auf Zeigerarithmetik innerhalb von Strukturen verzichtet (die nach dem Standard wiederum nicht definiert ist), ist diese Modellierung konsistent.

Eine weitere Vereinfachung ist, dass wir nur unstrukturierte Werte betrachten, d.h. keine Strukturen als Werte. Damit können wir nicht direkt Zuweisungen ganzer Strukturen modellieren (das ließe sich allerdings noch als eine Reihe von Zuweisungen der einzelnenen Komponenten modellieren), und insbesondere keine strukturwertige Funktionen (d.h. Funktionen, die eine Struktur zurückgeben). Damit ist der Zustand

$$\Sigma = \mathit{BaseLoc} \rightharpoonup (\mathbb{N} \times (\mathbb{N} \rightharpoonup \mathit{Val}))$$

Wir definieren die Funktionen zum Lesen und schreiben wie folgt:

$$\begin{aligned} \mathit{read} &: \Sigma \times \mathit{Loc} \rightharpoonup \mathit{Val} \\ \mathit{read}(\Sigma, (l, i)) &= (\Sigma l) i \\ \mathit{upd} &: \Sigma \times \mathit{Loc} \times \mathit{Val} \rightharpoonup \Sigma \\ \mathit{upd}(\Sigma, (l, i), v) &= \begin{cases} \Sigma(l \mapsto \emptyset(i \mapsto v)) & \Sigma l = \bot \\ \Sigma(l \mapsto (\Sigma l)(i \mapsto v)) & \text{sonst} \end{cases} \\ \mathit{fresh} &: \Sigma \times \mathit{Loc} \to \mathit{Bool} \\ \mathit{fresh}(\Sigma, (l, i)) &\Leftrightarrow \Sigma l = \bot \vee (\Sigma l) i = \bot \end{aligned}$$

Die vier Basis-Lemmata (3) bis (6) gelten auch analog für *read* und *upd*:

$$\begin{aligned} \mathit{read}(\mathit{upd}(\Sigma, l, v), l) &= v \\ \mathit{upd}(\mathit{upd}(\Sigma, l, v), l, w) &= \mathit{upd}(\Sigma, l, w) \\ l \neq m \longrightarrow \mathit{read}(\mathit{upd}(\Sigma, l, v), m) &= \mathit{read}(\Sigma, m) \\ l \neq m \longrightarrow \mathit{upd}(\mathit{upd}(\Sigma, l, v), m, w) &= \mathit{upd}(\mathit{upd}(\Sigma, m, w), l, v) \end{aligned}$$

Die Beweise erfordern einige Fallunterscheidungen und etwas Fleißarbeit.

Auf Adressen (*Loc*) können wir Hilfsfunktionen für den strukturierten Zugriff auf die dort gespeicherten Objekte definieren; diese entsprechen der Selektion von Feldern einer Struktur, und dem indizierten Zugriff in ein Feld. Zuerst benötigen wir ein Hilsfunktion, die den Offset einer Adresse erhöht.

$$\begin{aligned} \mathit{add\_offset} &: \mathit{Loc} \times \mathbb{N} \to \mathit{Loc} \\ \mathit{add\_offset}((l, n), i) &= (l, n + i) \end{aligned}$$

Die Funktion *rec_sel* und *rec_offset* geben für eine Struktur $Structi_1 : t_1, \ldots, i_n : t_n$ und Feldnamen j den Offset des Feldes j in der Struktur an. Falls j nicht in t enthalten ist, oder t keine Struktur ist, sind sie undefiniert.

$$rec\_sel : Loc \times T \times Idt \rightharpoonup Loc$$
$$rec\_offset : T \times Idt \times \rightharpoonup \mathbb{N}$$
$$rec\_sel(l, t, i) = add\_offsetl\, rec\_offset(t, i)$$
$$rec\_offset(\langle|\ i_1 : t_1, \ldots, i_n : t_n\ |\rangle j = \begin{cases} 0 & i_1 = j \\ size(t_1) + rec\_offset(\langle|\ i_2 : t_2, \ldots, i_n : t_n\ |\rangle) & i_1 \neq j \\ \bot & n = 0 \end{cases}$$
$$arr\_acc : Loc \times \mathbb{N} \times \mathbb{N} Loc$$
$$arr\_acc(l, s, i) = add\_offset(l, s \cdot i)$$

Wir definieren ferner noch folgende Projektionsfunktionen auf den Werten. Diese werden überall dort benötigt, wo wir nach der Auswertung eines Ausdrucks eine bestimmte Operation auf den Werten vornehmen wollen (beispielsweise in `*e` nach der Auswertung von `e` auf die Adresse projizieren, um an dieser Stelle den Speicher zu lesen).

$$p_{int} : Val \rightharpoonup \mathbb{Z}$$
$$p_{int}(z) = \begin{cases} z & z \in \mathbb{Z} \\ \bot & \text{sonst} \end{cases}$$
$$p_{real} : Val \rightharpoonup \mathbb{R}$$
$$p_{real}(r) = \begin{cases} r & r \in \mathbb{R} \\ \bot & \text{sonst} \end{cases}$$
$$p_{loc} : Val \rightharpoonup Loc$$
$$p_{loc}(l) = \begin{cases} l & l \in Loc \\ \bot & \text{sonst} \end{cases}$$

Ferner spezifizieren wir für arithemtische Typen Konversionsfunktionen und arithmetische Funktionen. Für je zwei arithmetische Typen $s, t \in T_{Arith}$ gibt es eine Konversionsfunktion (die wir hier nicht weiter definieren):

$$conv_{s,t} : s \rightarrow t$$

Die arithmetischen Funktionen sind partiell, da beispielsweise für vorzeichenbehaftete Ganzzahltypen Überlauf undefiniert ist. Für jeden arithmetischen Typ t gibt es:

$$add_t : t \times t \rightharpoonup t$$
$$sub_t : t \times t \rightharpoonup t$$
$$mult_t : t \times t \rightharpoonup t$$
$$div_t : t \times t \rightharpoonup t$$
$$mod_t : t \times t \rightharpoonup t$$

3.2 Die Semantik

Die Semantik ist definiert als eine Familie von semantischen Funktionen:

- Von C-Programmen:
$$[\![-]\!]_P : Prog \rightarrow Env$$

- Von Funktionen:

$$[\![-]\!]_f \quad : \mathit{FunDef} \to \mathit{Env} \to \mathit{Val\ list} \to \Sigma \to (\mathit{Val} \times \Sigma)$$

- Von Anweisungen:

$$[\![-]\!]_s \quad : \mathit{Stmt} \to \mathit{Env} \to \Sigma \to \Sigma$$

- Von (seiteneffektbehafteten) Ausdrücken:

$$[\![-]\!]_e \quad : \mathit{Stmt} \to \mathit{Env} \to \Sigma \to (\mathit{Val} \times \Sigma)$$

- Von L-Werten (keine Seiteneffekte!)

$$[\![-]\!]_{lv} \quad : \mathit{LVal} \to \mathit{Env} \to \Sigma \to \mathit{Loc}$$

3.2.1 L-Werte

$$\begin{aligned}
[\![x]\!]_{lv}\ \Gamma\ \Sigma =& \Gamma x (x \in \mathit{Idt}) \\
[\![e\texttt{[}i\texttt{]}]\!]_{lv}\ \Gamma\ \Sigma =& \mathit{arr_acc}([\![e]\!]_{lv}\ \Gamma\ \Sigma, s, j) \\
& \quad \text{wobei} \quad s = \mathit{size}(t), \Gamma \vdash e : \mathit{array}\ t\ n \text{ oder } \Gamma \vdash e : \mathit{ptr}\ t \\
& \qquad\quad j = p_{int}(\pi_1([\![\Gamma]\!]_e\ \Sigma\ i)) \\
[\![e\texttt{.}f]\!]_{lv}\ \Gamma\ \Sigma =& \mathit{rec_sel}([\![e]\!]_{lv}\ \Gamma\ \Sigma, t, f), \Gamma \vdash e : t \\
[\![e\texttt{->}f]\!]_{lv}\ \Gamma\ \Sigma =& [\![\texttt{*}(e\texttt{.}f)]\!]_{lv}\ \Gamma\ \Sigma \\
[\![\texttt{*}e]\!]_{lv}\ \Gamma\ \Sigma =& \mathit{read}(\Sigma, p_{loc}\pi_1([\![\Gamma]\!]_e\ \Sigma\ e))
\end{aligned}$$

3.2.2 Ausdrücke

$$\begin{aligned}
[\![\texttt{\&}e]\!]_e\ \Gamma\ \Sigma &= ([\![e]\!]_{lv}\ \Gamma\ \Sigma, \Sigma), e \in \mathit{LVal} \\
[\![e]\!]_e\ \Gamma\ \Sigma &= (\mathit{read}(\Sigma, [\![e]\!]_{lv}\ \Gamma\ \Sigma), \Sigma) \\
&\quad \text{wobei } e \in \mathit{LVal}, \Gamma \vdash e : t, t \neq \mathit{array}\ s\ n \\
&\quad \text{§6.3.2.1 (2), Konversion von L-Werten} \\
[\![e\texttt{++}]\!]_e\ \Gamma\ \Sigma &= (\mathit{read}(\Sigma, l), \mathit{upd}(\Sigma, l, \mathit{add}_1(\mathit{read}(\Sigma, l), 1))) \\
&\quad \text{wobei } \Gamma \vdash e : t, l = [\![e]\!]_{lv}\ \Gamma\ \Sigma \\
[\![e\ \texttt{*}\ f]\!]_e\ \Gamma\ \Sigma &= (\mathit{mult}_t(\mathit{conv}_{r,t}(v_1), \mathit{conv}_{s,t}(v_2)), \Sigma'') \\
&\quad \text{wobei} \quad [\![e]\!]_e\ \Gamma\ \Sigma = (v_1, \Sigma'), [\![f]\!]_e\ \Gamma\ \Sigma' = (v_2, \Sigma'') \\
&\qquad\quad \Gamma \vdash e : r, \Gamma \vdash f : s, \lhd_i\ rst \\
[\![e\ \texttt{+}\ f]\!]_e\ \Gamma\ \Sigma &= (\mathit{add}_t(\mathit{conv}_{r,t}(v_1), \mathit{conv}_{s,t}(v_2)), \Sigma'') \\
&\quad \text{wenn} \quad [\![e]\!]_e\ \Gamma\ \Sigma = (v_1, \Sigma'), [\![f]\!]_e\ \Gamma\ \Sigma' = (v_2, \Sigma'') \\
&\qquad\quad \Gamma \vdash e : r, \Gamma \vdash f : s, \lhd_i\ rst \\
[\![e\ \texttt{+}\ f]\!]_e\ \Gamma\ \Sigma &= (\mathit{arr_acc}(p_{loc}v_1, \mathit{sizet}, p_{int}v_2), \Sigma'') \\
&\quad \text{wenn} \quad [\![e]\!]_e\ \Gamma\ \Sigma = (v_1, \Sigma'), [\![f]\!]_e\ \Gamma\ \Sigma' = (v_2, \Sigma'') \\
&\qquad\quad \Gamma \vdash e : \mathit{ptr}\ t, \Gamma \vdash f : s, s \in T_{Int} \\
[\![e\ \texttt{+}\ f]\!]_e\ \Gamma\ \Sigma &= (\mathit{arr_acc}(p_{loc}v_2, \mathit{sizes}, p_{int}v_1), \Sigma'') \\
&\quad \text{wenn} \quad [\![e]\!]_e\ \Gamma\ \Sigma = (v_1, \Sigma'), [\![f]\!]_e\ \Gamma\ \Sigma' = (v_2, \Sigma'') \\
&\qquad\quad \Gamma \vdash e : s, \Gamma \vdash f : \mathit{ptr}\ t, \in T_{Int}
\end{aligned}$$

Hier fehlen:

- Die Semantik für Präinkrement, und für Post- und Prädekrement;

- Die Semantik für Division und Modulus;
- Die Semantik für Subtraktion (enthält noch zusätzlich den Fall, dass zwei Zeiger auf denselben Basistyp voneinander subtrahiert werden können, vgl. Regel SUBTRAKTION-ZEIGER oben).

$$
\begin{array}{rcl}
[\![e \texttt{ == } f]\!]_e\ \Gamma\ \Sigma & = & \left\{ \begin{array}{ll} (0,\Sigma_1) & conv_{s,t}(v_1) \neq conv_{r,t}(v_2) \\ (1,\Sigma_2) & conv_{s,t}(v_1) = conv_{r,t}(v_2) \end{array} \right. \\
 & & \textsf{wenn}\ \ \Gamma \vdash e : s, \Gamma \vdash f : r, \lhd_a\ s, rt \\
 & & \quad [\![e]\!]_e\ \Gamma\ \Sigma = (v_1,\Sigma_1), [\![f]\!]_e\ \Gamma\ \Sigma_1 = (v_2,\Sigma_2) \\
[\![e \texttt{ == } f]\!]_e\ \Gamma\ \Sigma & = & \left\{ \begin{array}{ll} (0,\Sigma_1) & v_1 \neq v_2 \\ (1,\Sigma_2) & v_1 = v_2 \end{array} \right. \\
 & & \textsf{wenn}\ \ \Gamma \vdash e : ptr\ t, \Gamma \vdash f : ptr\ s \\
 & & \quad [\![e]\!]_e\ \Gamma\ \Sigma = (v_1,\Sigma_1), [\![f]\!]_e\ \Gamma\ \Sigma_1 = (v_2,\Sigma_2) \\
[\![e \texttt{ < } f]\!]_e\ \Gamma\ \Sigma & = & \left\{ \begin{array}{ll} (0,\Sigma_1) & conv_{s,t}(v_1) < conv_{r,t}(v_2) \\ (1,\Sigma_2) & conv_{s,t}(v_1) \geq conv_{r,t}(v_2) \end{array} \right. \\
 & & \textsf{wenn}\ \ \Gamma \vdash e : s, \Gamma \vdash f : r, \lhd_a\ s, rt \\
 & & \quad [\![e]\!]_e\ \Gamma\ \Sigma = (v_1,\Sigma_1), [\![f]\!]_e\ \Gamma\ \Sigma_1 = (v_2,\Sigma_2) \\
[\![e \texttt{ < } f]\!]_e\ \Gamma\ \Sigma & = & \left\{ \begin{array}{ll} (1,\Sigma_1) & \pi_1(v_1) = \pi_1(v_2) \wedge \pi_2(v_1) < \pi_2(v_2) \\ (0,\Sigma_2) & \pi_1(v_1) = \pi_1(v_2) \wedge \pi_2(v_1) \geq \pi_2(v_2) \\ \bot & \pi_1(v_1) \neq \pi_1(v_2) \end{array} \right. \\
 & & \textsf{wenn}\ \ \Gamma \vdash e : ptr\ t, \Gamma \vdash f : ptr\ s \\
 & & \quad [\![e]\!]_e\ \Gamma\ \Sigma = (v_1,\Sigma_1), [\![f]\!]_e\ \Gamma\ \Sigma_1 = (v_2,\Sigma_2)
\end{array}
$$

- Es fehlen noch Ungleichheit (`!=`) und die anderen Relationen (`<=`, `>`, `>=`).
- Bei der Semantik des Kleineroperators (und der anderen Relationen) auf Zeigern ist der Vergleich nur dann definiert (d.h. liefert 1 wahr oder 0 falsch), wenn beide Zeiger auf dasselbe Objekt zeigen, d.h. ihre *BaseLoc* ist gleich. Das entspricht für $l, m \in Loc$ der Gleichung $\pi_1(l) = \pi_1(m)$.

$$
\begin{array}{rcl}
[\![e \texttt{ \&\& } f]\!]_e\ \Gamma\ \Sigma & = & \left\{ \begin{array}{ll} (0,\Sigma_1) & v_1 = 0 \\ (0,\Sigma_2) & v_1 \neq 0, v_2 = 0 \\ (1,\Sigma_2) & v_1 \neq 0, v_2 \neq 0 \end{array} \right. \\
 & & \textsf{wobei}\ (v_1,\Sigma_1)[\![e]\!]_e\ \Gamma\ \Sigma, (v_2,\Sigma_2) = [\![\Gamma]\!]_e\ \Sigma_1\ f \\
[\![e \texttt{ || } f]\!]_e\ \Gamma\ \Sigma & = & \left\{ \begin{array}{ll} (1,\Sigma_1) & v_1 \neq 0 \\ (1,\Sigma_2) & v_1 = 0, v_2 \neq 0 \\ (0,\Sigma_2) & v_1 = 0, v_2 = 0 \end{array} \right. \\
 & & \textsf{wobei}\ (v_1,\Sigma_1)[\![e]\!]_e\ \Gamma\ \Sigma, (v_2,\Sigma_2) = [\![\Gamma]\!]_e\ \Sigma_1\ f
\end{array}
$$

- Hier fehlen die Regeln für `&`, `|` und `^`.
- Die Regeln für logische Konjunktion und Disjunktion stellen die Nicht-Striktheit auf der rechten Seite sicher, d.h. wenn die Auswertung des ersten Argumentes 0 (1) ergibt, dann wird das andere Argument der Konjunktion (Disjunktion) nicht mehr ausgewertet.

$$
\begin{array}{rcl}
[\![e \texttt{ = } f]\!]_e\ \Gamma\ \Sigma & = & (conv_{t,s}(v), upd(\Sigma_2, l, conv_{t,s}(v))) \\
 & & \textsf{wobei}\ \ [\![e]\!]_{lv}\ \Gamma\ \Sigma = (l,\Sigma_1), [\![\Gamma]\!]_e\ \Sigma_1\ f = (v,\Sigma_2) \\
 & & \quad \Gamma \vdash e : s, \Gamma \vdash f : t
\end{array}
$$

- Nicht modelliert ist hier, dass l ein modifzierbarer L-Wert sein muss (d.h. der Typ darf nicht mit `const` qualifiziert sein).
- Es fehlt noch die Fallunterscheidung (nicht ganz trivial).

3.2.3 Anweisungen

Für die Semantik $[\![-]\!]_s$ von Anweisungen verweisen wir auf Standardtexte [2]; hier ist C nichts besonderes.

3.2.4 Funktionsdefinitionen und Blöcke

Wir definieren erst eine einfachere abstrakte Syntax für Deklarationen und Blöcke als die im Standard:

$$\begin{aligned} \mathit{FunDef} &= \mathit{Idt}\ \mathit{ParamDecls}\ \mathit{Blk} \\ \mathit{ParamDecls} &= (\mathit{Idt}\ \mathit{type})\ \mathit{ParamDecls} | \varepsilon \\ \mathit{Blk} &= (\mathit{Idt}\ \mathit{type})\ \mathit{Blk} | \mathit{Body} \\ \mathit{Body} &= \mathit{Stmt}\ (\texttt{return}\ \mathit{Expr})^+ \end{aligned}$$

Eine Funktionsdefinition besteht aus einem Bezeichner, einer Liste von Parameterdeklarationen, und einem Block. Eine Parameterdeklaration besteht aus einem Bezeichner und einem Typ. Ein Block ist eine Liste von (lokalen) Variablendeklarationen, bestehend aus einer Variable und einem Typ, sowie einem Rumpf, der aus einem Statement, gefolgt von einer optionalen Rückgabeanweisung, besteht.
Die Definition von $[\![-]\!]_f$ benötigt folgende Hilfsfunktionen:

$$\begin{aligned} [\![-]\!]_{ps} &: \mathit{ParamDecls} \to \mathit{Env} \to \mathit{Val\ list} \to \Sigma \to (\mathit{Val} \times \Sigma) \\ [\![-]\!]_{blk} &: \mathit{Blk} \to \mathit{Env} \to \mathit{Val\ list} \to \Sigma \to (\mathit{Val} \times \Sigma) \end{aligned}$$

Vereinfachung: In dieser Semantik sind eine Reihe von Vereinfachungen:

- Wir machen hier die stark vereinfachende Annahme, dass der Rückgabewert der Funktion nur einmal am Ende berechnet wird. Das ist beispielsweise konsistent mit dem MISRA-Standard (kann aber zu unnötig verschachtelten Programmcode führen). Wir machen die Annahme an dieser Stelle, um nicht über Ausnahmenbehandlung reden zu müssen, was zwar Stand der Technik ist, aber wiederum nicht C-spezifisch und damit nicht Fokus der Veranstaltung.
- Wir berücksichtigen keine Initialisierer.
- Die Definition von $[\![-]\!]_{blk}$ ist ungenau, und berücksichtigt nicht den Fall, dass keine Rückgabenanweisung am Ende des Rumpfes vorliegt.

In der folgenden Semantik wird der Hilbert-Operator ε verwendet. Dieser beschreibt für ein Prädikat P über X ein beliebiges $x \in X$, so dass $P(x)$ gilt. Wir verwenden ihn hier für die Initialisierung: der Initialwert einer Variablen ist ein beliebiger Wert aus dem Wertebereich des Typs der Variablen.

$$
\begin{array}{rcl}
[\![f\ ps\ blk]\!]_f\ \Gamma\ args\ \Sigma & = & [\![ps\ blk]\!]_{ps}\ \Gamma\ args\ \Sigma \\
[\![(i\ t)\ ps\ blk]\!]_{ps}\ \Gamma\ args\ \Sigma & = & [\![ps\ blk]\!]_{ps}\ \Gamma, (i \mapsto l)\ hd(args)\ (upd(\Sigma, l, hd(args))) \\
 & & \text{wobei } fresh(\Sigma, l) \\
[\![\varepsilon\ blk]\!]_{ps}\ \Gamma\ args\ \Sigma & = & [\![blk]\!]_{blk}\ \Gamma\ \Sigma \\
[\![(i\ t)\ blk]\!]_{blk}\ \Gamma\ \Sigma & = & [\![blk]\!]_{blk}\ \Gamma, (i \mapsto l)\ upd(\Sigma, l, \varepsilon x.x \in val(t)) \\
 & & \text{wobei } fresh(\Sigma, l) \\
[\![s(\texttt{return}e)]\!]_{blk}\ \Gamma\ \Sigma & = & [\![e]\!]_e\ \Gamma\ ([\![s]\!]_s\ \Gamma\ \Sigma)
\end{array}
$$

3.2.5 Funktionsaufrufe

Damit können wir jetzt auch Funktionsaufrufe modellieren. Wir benötigen eine Hilfsfunktion, die Liste von Ausdrücken (konkret die Liste der Argumente der Funktion) auswertet:

$$
\begin{array}{rcl}
[\![-]\!]_{es} & : & ExprList \to Env \to \Sigma \to (Val\ list, \Sigma) \\
[\![e,\ as]\!]_{es}\ \Gamma\ \Sigma & = & (v \# vs, \Sigma_2) \\
 & & \text{wobei } (v, \Sigma_1) = [\![e]\!]_e\ \Gamma\ \Sigma, (vs, \Sigma_2) = [\![as]\!]_{es}\ \Gamma\ \Sigma_2 \\
[\![\varepsilon]\!]_{es}\ \Gamma\ \Sigma & = & ([], \Sigma) \\
[\![f(as)]\!]_e\ \Gamma\ \Sigma & = & (\Gamma f) a \Sigma', (a, \Sigma') = [\![as]\!]_{es}\ \Gamma\ \Sigma
\end{array}
$$

4 Ein Hoare-Kalkül für C

Probleme mit dem herkömlichen Hoare-Kalkül: Referenzen und Seiteneffekte. Daher:

- Nachbedingung kann auch über Ergebnis reden
- Regel über Umgebung parametrisiert

Definition: Hoare-Tripel für seitenffektbehaftete Ausdrücke:

$$
\begin{array}{l}
- \vdash_e \{-\} - \{-\} : Env \to (\Sigma \to Bool) \to Expr \to (Val \times \Sigma \to Bool) \to Bool \\
\Gamma \vdash_e \{P\} e \{Q\} \equiv (\forall S.P\ S \longrightarrow (Q\ ([\![e]\!]_e\ \Gamma\ S)))
\end{array}
$$

Die Gültigkeit kann durch folgende Regeln hergeleitet werden, deren Korrektheit als abgeleitete Theoreme bewiesen werden kann.

$$
\frac{n \in Lit}{\Gamma \vdash_e \{Q[\![n]\!]lit\} n \{Q\}} \qquad \text{(Literale)}
$$

$$
\frac{l \in LVal}{\Gamma \vdash_e \{Q(read(\Sigma, [\![l]\!]_{lv}\ \Gamma\ \Sigma))\} l \{Q\}} \qquad \text{(L-Werte)}
$$

Anmerkung: Das setzt voraus, dass der L-Werte keinen Seiteneffekte hat (das war schon eine Voraussetzung bei der Semantik).

$$
\frac{l \in LVal}{\begin{array}{l}\Gamma \vdash_e \{\lambda\Sigma.\, Q(v+1)(upd(\Sigma, m, v+1))\} {+}{+} l \{Q\} \\ \text{wobei } \begin{array}{rcl} m & = & [\![l]\!]_{lv}\ \Gamma\ \Sigma \\ v & = & read(\Sigma, m) \end{array}\end{array}} \qquad \text{(Präinkrement)}
$$

$$\frac{l \in LVal}{\begin{array}{l}\Gamma \vdash_e \{\lambda\Sigma.\, Q(v-1)(upd(\Sigma, m, v-1))\}\texttt{--}l\{Q\} \\ \textsf{wobei} \quad m = [\![l]\!]_{lv}\ \Gamma\ \Sigma \\ \qquad\quad v = read(\Sigma, m)\end{array}} \qquad \text{(Prädekrement)}$$

$$\frac{l \in LVal}{\begin{array}{l}\Gamma \vdash_e \{\lambda\Sigma.\, Q\ v\ (upd(\Sigma, m, v+1))\}l\texttt{++}\{Q \\ \textsf{wobei} \quad m = [\![l]\!]_{lv}\ \Gamma\ \Sigma \\ \qquad\quad v = read(\Sigma, m)\end{array}} \qquad \text{(Postinkrement)}$$

$$\frac{l \in LVal}{\begin{array}{l}\Gamma \vdash_e \{\lambda\Sigma.\, Q\ v\ (upd(\Sigma, m, v-1))\}l\texttt{--}\{Q\} \\ \textsf{wobei} \quad m = [\![l]\!]_{lv}\ \Gamma\ \Sigma \\ \qquad\quad v = read(\Sigma, m)\end{array}} \qquad \text{(Postdekrement)}$$

Anmerkung: Die vier Regeln unterscheiden sich nur in den Argumenten von Q in der Vorbedingung. Die Notation $v+1$ und $v-1$ ist etwas unpräzise, genauer muss es heißen $add_t(v, 1)$ für $Gamma \vdash v : t$.

$$\frac{\Gamma \vdash_e \{P\}e\{\lambda v.\, Q(-v)\}}{\Gamma \vdash_e \{P\}\texttt{-}e\{Q\}} \qquad \text{(Unäres Minus)}$$

$$\frac{l \in LVal}{\Gamma \vdash_e \{Q([\![l]\!]_{lv}\ \Gamma\ \Sigma)\}\texttt{\&}l\{Q\}} \qquad \text{(Referenz)}$$

Anmerkung: Vergl. mit (L-Werte).

$$\frac{\begin{array}{c}\Gamma \vdash e : r \qquad \Gamma \vdash f : s \qquad (r, s) \rhd_a t \\ \Gamma \vdash_e \{P\}e\{R\} \qquad \forall n.\, \Gamma \vdash_e \{R\ n\}f\{\lambda v.\, Q\ (add_t(conv_{r,t}(n), conv_{s,t}(v)))\}\end{array}}{\Gamma \vdash_e \{P\}e\texttt{+}f\{Q\}} \qquad \text{(Addition-1)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : s \qquad \Gamma \vdash f : ptr\ t \\ \Gamma \vdash_e \{P\}e\{R\} \qquad \forall n.\, \Gamma \vdash_e \{R\ n\}f\{\lambda v.\, Q\ (arr\_acc(v, t, n))\}\end{array}}{\Gamma \vdash_e \{P\}e\texttt{+}f\{Q\}} \qquad \text{(Addition-2a)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : ptr\ s \qquad \Gamma \vdash f : t \\ \Gamma \vdash_e \{P\}e\{R\} \qquad \forall n.\Gamma \vdash_e \{R\ n\}f\{\lambda v.\, Q\ (arr\_acc(v, s, n))\}\end{array}}{\Gamma \vdash_e \{P\}e\texttt{+}f\{Q\}} \qquad \text{(Addition-2b)}$$

Anmerkung: Ähnliche Regeln für Subtraktion, und andere zweistellige Operatoren: Multiplikation, Division, Modulo,`&`, `|`, `^`, ...

$$\frac{\begin{array}{c}\Gamma \vdash e : r \qquad \Gamma \vdash f : s \qquad (r, s) \rhd_a t \\ \Gamma \vdash_e \{P\}e\{R\} \qquad \forall n.\, \Gamma \vdash_e \{R\ n\}f\{\lambda v.\, Q(conv_{r,t}(n) = conv_{s,t}(v))\}\end{array}}{\Gamma \vdash_e \{P\}e\texttt{==}f\{Q\}} \qquad \text{(Gleichheit-1)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : ptr\ t \qquad \Gamma \vdash f : ptr\ s \\ \Gamma \vdash_e \{P\}e\{R\} \qquad \forall n.\, \Gamma \vdash_e \{R\ n\}f\{\lambda v.\, Q(n = v)\}\end{array}}{\Gamma \vdash_e \{P\}e == f\{Q\}} \quad \text{(Gleichheit-2)}$$

Anmerkung: Ähnliche Regeln für die anderen Relationen.

$$\frac{\Gamma \vdash_e \{P\}e\{Q\ False\}}{\Gamma \vdash_e \{P\}e \,\&\&\, f\{Q\}} \quad \text{(Konjunktion-1)}$$

$$\frac{\Gamma \vdash_e \{P\}e\{R\} \qquad \forall b.\, \Gamma \vdash_e \{R\ b\}f\{\lambda v.\, Q(n \wedge v)\}}{\Gamma \vdash_e \{P\}e \,\&\&\, f\{Q\}} \quad \text{(Konjunktion-2)}$$

$$\frac{\Gamma \vdash_e \{P\}e\{\lambda v\, \Sigma.\, Q\ v\ (upd(\Sigma, [\![x]\!]_{lv}\ \Gamma\ \Sigma, v))\}}{\Gamma \vdash_e \{P\}x = e\{Q\}} \quad \text{(Zuweisung)}$$

$$\frac{\Gamma \vdash_e \{P\}c\{Q\} \qquad \Gamma \vdash_e \{Q\ True\}e\{R\} \qquad \Gamma \vdash_e \{Q\ False\}f\{R\}}{\Gamma \vdash_e \{P\}c\,?\,e : f\{R\}} \quad \text{(Konditional)}$$

Literatur

[1] Peter van der Linden. *Expert C Programming: Deep C Secrets*. SunSoft Press. Prentice-Hall, 1994.

[2] Glynn Winskel. *The Formal Semantics of Programming Langauges*. Foundations of Computing Series. The MIT Press, 1993.