Programmiersprachen

Vorlesung 2 vom 19.10.23

Einfache Ausdrücke

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Wo sind wir?

Ausdrücke

Ausdrücke und Anweisungen

- Viele Sprachen unterscheiden **Ausdrücke** und **Anweisungen**
  - Ausdrücke bezeichnen die zu manipulierenden **Werte**
  - Anweisungen beschreiben **Kontrollfluß**
  - **Imperatives** Konstrukt: Programm ist abstrakte Sicht auf Speichermanipulation
- Funktionale Sprachen, logische Sprachen (**deklarative**) uvm. haben diese Unterscheidung **nicht**.

Werte

- Was sind Werte?
  "Any entity that can be manipulated by a program."
  - **Primitive** Werte:
    Booleans, ganze Zahlen, Fließkommazahlen, Adressen (Pointer, Referenzen)
  - **Zusammengesetzte** Werte (composite values):
    Strukturen, Felder, Listen, Objekte, ...
- Semantisch gesehen:
  - Abstraktionen über dem Speicherinhalt
  - Nichtreduzierbare Ausdrücke

Typen

- Was sind Typen?
  - "A set of values." (Mengen von Werten)
  - Durch die Operationen darauf charakterisiert. z.B. $\{13, \mathit{Monday}, \mathit{true}\}$ ist kein Typ.
    — kann man drüber streiten
- Arten von Typen:
  - **Primitive** Typen (primitive types)
  - **Zusammengesetzte** Typen (composite types)
- Typüberprüfung und Typsysteme

Vorgegebene Primitive Typen

- Anwendungsabhängig: COBOL vs. FORTRAN; BCPL vs. C vs. TEX
- Ganze Zahlen (feste Wortbreite):
  - Java: `int`
  - C: `char`, `short`, `int`, `long`, `long long`, etc.
  - Rust: `i8`,...,`i128`, `u8`,...,`u128`, `isize`, `usize`
  - Haskell: `Int`
- C und Rust haben `signed` und `unsigned` ($\mathbb{Z}$ und $\mathbb{N}$)
- Vorzeichen meist mit **Zweierkomplement.**

Vorgegebene Primitive Typen

- Beliebig große ganze Zahlen:
  - `Integer`, `Natural` in Haskell
  - `BigInteger` in Java
  - Basiert auf Gnu MP Bücherei
- Booleans:
  - In C **kein** expliziter Typ (`bool\_t` ist `int`)
- Fließkommazahlen:
  - `float` und `double` (in Rust: `f32` und `f64`).
  - Meist nach IEEE 754

Übung 1.1: Welche Wortbreite haben ganze Zahlen in C, Java, Python, Haskell?

Selbstdefinierte Primitive Typen

- C, Java, Haskell erlauben **Aufzählungen:**

```
enum colour {red, green, blue}
```

- In C nur syntaktischer Zucker für `int`, kein separater Typ.
- In Java: Klasse mit konstanter Anzahl von Objekten (s. hier)
- In Haskell und Rust: algebraischer Typ mit ausschließlich konstanten Konstruktoren.

```
data Colour = Red | Green | Blue
```

```
enum Colour  { Red, Green, Blue }
```


Zeichenketten

- Zeichenketten (Strings) spielen **meist** eine Sonderrolle
- Konzeptionell: Array oder Liste von Zeichenketten, *e.g.* C und Haskell:

```
char txt1[] = "Foo";
char txt2[4] = {'F', 'o', 'o', 0};
```

```
type String = [Char]
```

- Aus Effizienzgründen: **Unveränderliche** Strings
  - Java und Python
  - `bytestring` in Haskell

Auswertung

Ausdrücke

- Wir beschreiben die Semantik mit Hilfe einer **vereinfachten Sprache**:

$$\begin{aligned} e ::= \mathbb{Z} \mid \textbf{Idt} \mid \mathit{true} \mid \mathit{false} \\ \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \\ \mid e_1 == e_2 \mid e_1 < e_2 \\ \mid !\, e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \end{aligned}$$

- Dieses ist **abstrakte Syntax**. Terme können als **Bäume** repräsentiert werden.

Strukturelle Operationale Semantik

- Induktiv definiert durch **Regeln** der Form

$$\frac{\langle a', t' \rangle \to b' \qquad \phi(a')}{\langle a, t \rangle \to b}$$

- t ist ein Baum der Tiefe 1 (ein oberstes Symbol mit variablen Kindern)
- a sind Eingabedaten (*e.g.* der Zustand), b Rückgabe (*e.g.* ein Wert)
- Die Anwendung einer Regel entspricht einem **Zustandsübergang**

Zustände

Zustände sind **partielle, endliche Abbildungen** (finite partial maps) repräsentiert durch **rechtseindeutige** Relationen

$$f : X \rightharpoonup A \subseteq X \times A \text{ so dass } \forall x, a, b.\, (x, a) \in f \wedge (x, b) \in f \implies a = b$$

Notation:

- $\langle x \mapsto a, y \mapsto b, z \mapsto c \rangle$ für $\{(x, a), (y, b), (z, c)\}$
- $f(x)$ für den Wert von x in f (*lookup*)
- $f(x) = \bot$ wenn x nicht in f (*undefined*) und $\mathrm{def}(f(x))$ für $(x, y) \in f$ (*defined*)
- $f \setminus x$ um x aus f zu entfernen (*remove*)
- $f[x \mapsto n]$ für den **Update** an der Stelle x mit dem Wert n.

Regeln der Operationalen Semantik

Ein Ausdruck e wertet unter Zustand σ zu einer ganzen Zahl n oder einen boolschen Wert b aus.

$$e ::= \mathbb{Z} \mid \textbf{Idt} \mid \mathit{true} \mid \mathit{false} \mid \ldots \qquad \langle e, \sigma \rangle \to_{Exp} n \mid b$$

Regeln:

$$\frac{i \in \mathbb{Z}}{\langle i, \sigma \rangle \to_{Exp} i} \qquad \frac{b \in \mathbb{B}}{\langle b, \sigma \rangle \to_{Exp} b} \qquad \frac{x \in \textbf{Idt}, x \in \mathrm{dom}(\sigma), \sigma(x) = v}{\langle x, \sigma \rangle \to_{Exp} v}$$

Operationale Semantik: Arithmetische Ausdrücke

$$e ::= \ldots \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \mid \ldots \qquad \langle e, \sigma \rangle \to_{Exp} n$$

Regeln:

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}}{\langle e_1 + e_2, \sigma \rangle \to_{Exp} n_1 + n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}}{\langle e_1 - e_2, \sigma \rangle \to_{Exp} n_1 - n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}}{\langle e_1 * e_2, \sigma \rangle \to_{Exp} n_1 \cdot n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_2 \neq 0}{\langle e_1 / e_2, \sigma \rangle \to_{Exp} n_1 \div n_2}$$

Operationale Semantik: Prädikate

$$e ::= \ldots \mid e_1 == e_2 \mid e_1 < e_2 \mid \ldots \quad \langle e, \sigma \rangle \to_{Exp} b$$

Regeln:

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 = n_2}{\langle e_1 == e_2, \sigma \rangle \to_{Lexp} true}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 \neq n_2}{\langle e_1 == e_2, \sigma \rangle \to_{Lexp} false}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 < n_2}{\langle e_1 < e_2, \sigma \rangle \to_{Lexp} true}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Exp} n_1 \qquad \langle e_2, \sigma \rangle \to_{Exp} n_2 \qquad n_i \in \mathbb{Z}, n_1 \geq n_2}{\langle e_1 < e_2, \sigma \rangle \to_{Lexp} false}$$

Operationale Semantik: Konnektive

$$e ::= \ldots \mid ! e \mid e_1 \,\&\&\, e_2 \mid e_1 \,||\, e_2 \quad \langle e, \sigma \rangle \to_{Exp} b$$

Regeln:

$$\frac{\langle e, \sigma \rangle \to_{Lexp} true}{\langle ! e, \sigma \rangle \to_{Lexp} false} \qquad \frac{\langle e, \sigma \rangle \to_{Lexp} false}{\langle ! e, \sigma \rangle \to_{Lexp} true}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Lexp} false}{\langle e_1 \,\&\&\, e_2, \sigma \rangle \to_{Lexp} false} \qquad \frac{\langle e_1, \sigma \rangle \to_{Lexp} true \qquad \langle e_2, \sigma \rangle \to_{Lexp} t}{\langle e_1 \,\&\&\, e_2, \sigma \rangle \to_{Lexp} t}$$

$$\frac{\langle e_1, \sigma \rangle \to_{Lexp} true}{\langle e_1 \,||\, e_2, \sigma \rangle \to_{Lexp} true} \qquad \frac{\langle e_1, \sigma \rangle \to_{Lexp} false \qquad \langle e_2, \sigma \rangle \to_{Lexp} t}{\langle e_1 \,||\, e_2, \sigma \rangle \to_{Lexp} t}$$

Zusammenfassung

- Ausdrücke werden im Kontext eines **Zustands** zu **Werten** ausgewertet.
- Zustände sind partielle endliche Abbildungen.
- Strukturelle Operationale Semantik beschreibt diese Auswertung **mathematisch**.
- Nicht alle Ausdrücke lassen sich auswerten.
- Typen versuchen diese Undefiniertheit im Vorfeld auszuschließen.