Funktionales Programmieren

Teil 11

Carl Philipp Reh

Universität Siegen

12. August 2020

Semantik

Für eine Umgebung η mit Definitionsbereich $\{x_1, \ldots, x_n\}$, wobei $\eta(x_i) = d_i$ für $1 \leq i \leq n$, schreiben wir einfach $[x_1/d_1, \ldots, x_n/d_n]$.

Zu einem Ausdruck $e \in \text{Exp}$ und einer Umgebung $\eta\colon \text{Var} \to \text{Dom}$ können wir nun *die Semantik* $[\![\mathrm{e}]\!]_\eta \in \text{Dom}$ definieren, falls $\eta(x)$ für alle $x \in \text{free}(e)$ definiert ist. Dafür machen wir eine Fallunterscheidung über die möglichen Ausdrücke.

Für Variablen $x \in \text{Var}$ sei $[\![\mathrm{x}]\!]_\eta = \eta(x)$. Das heißt, dass Variablen einfach in der Umgebung nachgeschlagen werden. Hier ist natürlich wichtig, dass η auf x definiert ist.

Für Integerwerte $i \in \mathbb{Z}$ sei $[\![\mathrm{i}]\!]_\eta = i$.

Für die Applikation von zwei Ausdrücken, also das Übergeben eines Parameters an eine Funktion, sei $[\![(\mathrm{e}_1\mathrm{e}_2)]\!]_\eta = [\![\mathrm{e}_1]\!]_\eta([\![\mathrm{e}_2]\!]_\eta)$, falls $[\![\mathrm{e}_1]\!]_\eta\colon \text{Dom} \to \text{Dom}$. Ansonsten ist $[\![(\mathrm{e}_1\mathrm{e}_2)]\!]_\eta = \bot$. Letzteres bedeutet, dass $[\![\mathrm{e}_1]\!]_\eta$ keine Funktion ist, zum Beispiel bei $[\![(1\ 2)]\!]_\eta$.

Semantik

Für einen Konstruktor $\underline{\mathrm{C}} \in \mathrm{Con}_0$ sei $[\![\underline{\mathrm{C}}]\!]_\eta = \underline{\mathrm{C}}$. Zum Beispiel gilt $[\![\underline{\mathrm{True}}]\!]_\eta = \underline{\mathrm{True}}$.
Für einen Konstruktor $\underline{\mathrm{C}} \in \mathrm{Con}_n$ für $n > 0$ sei $[\![\underline{\mathrm{C}}]\!] = f$ die Funktion $f\colon \underbrace{\mathrm{Dom} \to \cdots \to \mathrm{Dom}}_{n \text{ mal}} \to \mathrm{Dom}$ mit $f(d_1) \cdots (d_n) = (\underline{\mathrm{C}}, d_1, \ldots, d_n)$. Das heißt, dass $[\![\underline{\mathrm{C}}]\!]$ eine n-stellige Funktion ist, die ein Element aus $\mathrm{Constructors}_n$ liefert. Zum Beispiel gilt $[\![\underline{\mathrm{Succ}}]\!]_\eta\colon \mathrm{Dom} \to \mathrm{Dom}$ mit $[\![\underline{\mathrm{Succ}}]\!]_\eta(x) = (\underline{\mathrm{Succ}}, x)$.

Für Lambda-Ausdrücke sei $[\![\backslash \mathrm{x} -> \mathrm{e}]\!]_\eta = f$, wobei $f\colon \mathrm{Dom} \to \mathrm{Dom}$ die Funktion ist mit $f(d) = [\![\mathrm{e}]\!]_{\eta \oplus [x/d]}$. Wir müssen also den Wert für den Parameter x in die Umgebung η eintragen. Zum Beispiel gilt $[\![\backslash \mathrm{x} -> (\underline{\mathrm{Succ}}\ \mathrm{x})]\!]_\eta = f$, wobei $f(d) = [\![(\underline{\mathrm{Succ}}\ \mathrm{x})]\!]_{\eta \oplus [x/d]} = (\underline{\mathrm{Succ}}, d)$.

Semantik

In unserer vereinfachten Sprache sind Let-Ausdrücke der einzige Weg, rekursive Funktionen zu definieren. Bei let $x = e_1$ in e_2 wird x sowohl in e_1 als auch in e_2 gebunden. Zum Beispiel sei

$$e := \text{let } \mathrm{fact} = \backslash x \text{ -> case } x \leq 0 \text{ of } \{\underline{\mathrm{True}} \text{ -> } 1 \,;\, \underline{\mathrm{False}} \text{ -> } x \cdot \mathrm{fact}\ (x-1)\} \text{ in } \mathrm{fact}\ 2$$

Um `fact` eine Semantik im Ausdruck `fact 2` zu geben, bilden wir den kleinsten Fixpunkt der Funktion $f : \mathrm{Dom} \to \mathrm{Dom}$ mit

$$f(d) = [\![\backslash x \text{ -> case } x \leq 0 \text{ of } \{\underline{\mathrm{True}} \text{ -> } 1 \,;\, \underline{\mathrm{False}} \text{ -> } x \cdot \mathrm{fact}\ (x-1)\}]\!]_{\eta \oplus [\mathrm{fact}/d]}.$$

Dann definieren wir $[\![\mathrm{e}]\!]_\eta = [\![\mathrm{fact}\ 2]\!]_{\eta \oplus [\mathrm{fact}/\mu f]}$.

Allgemein sei $[\![\text{let } \mathrm{x} = \mathrm{e}_1 \text{ in } \mathrm{e}_2]\!]_\eta = [\![\mathrm{e}_2]\!]_{\eta \oplus [x/\mu f]}$, wobei $f : \mathrm{Dom} \to \mathrm{Dom}$ mit $f(d) = [\![\mathrm{e}_1]\!]_{\eta \oplus [x/d]}$.

Semantik

Sei $e = \text{case } e' \text{ of } \{p_1 \texttt{ -> } e_1 \texttt{ ; } \cdots \texttt{ ; } p_n \texttt{ -> } e_n\}$. Hier gehen wir wie folgt vor: Haskell versucht zunächst, e' so weit auszuwerten, bis klar ist, welcher Konstruktor angewandt wurde. In unserer Semantik bedeutet das, dass $[\![\mathrm{e}']\!]_\eta = (\underline{\mathrm{C}}, d_1, \ldots, d_m)$ sein muss, wobei $\underline{\mathrm{C}} \in \mathrm{Con}_m$ für ein $m \in \mathbb{N}$ und $d_1, \ldots, d_m \in \mathrm{Dom}$. Ist dies nicht der Fall, dann ist $[\![\mathrm{e}]\!]_\eta = \bot$. Man beachte, dass es nicht vorkommen kann, dass $[\![\mathrm{e}']\!]_\eta = (\underline{\mathrm{C}}, d_1, \ldots, d_m)$, aber $\underline{\mathrm{C}} \in \mathrm{Con}_n$ mit $n \neq m$. Das liegt daran, dass $[\![\underline{\mathrm{C}}]\!]$ für $\underline{\mathrm{C}} \in \mathrm{Con}_m$ immer genau m Argumente erwartet.

Der nächste Schritt ist, den ersten Pattern (von links nach rechts) rauszusuchen, der $\underline{\mathrm{C}}$ verwendet. Hierbei ist es wichtig, dass der Pattern genau m Variablen enthält. Gibt es keinen solchen Pattern, so ist auch $[\![\mathrm{e}]\!]_\eta = \bot$. Ansonsten sei $p_i = \underline{\mathrm{C}}\ x_1\ \ldots\ x_m$ (wobei $1 \leq i \leq n$) dieser erste Pattern. Dann müssen wir d_j für x_j in die Umgebung einsetzen (für $1 \leq j \leq m$) und darunter e_i auswerten.

Semantik

Wir schreiben $C(p_i)$ für den Konstruktor, der in p_i steht, also $C(p_i) = \underline{\mathrm{C_i}}$, falls $p_i = (\underline{\mathrm{C_i}}, x_1, \ldots, x_m)$ für $x_1, \ldots, x_m \in \mathrm{Var}$ und $\underline{\mathrm{C_i}} \in \mathrm{Con}_m$. Dann definieren wir die Semantik von e als

$$[\![\mathrm{e}]\!]_\eta = \begin{cases} [\![\mathrm{e_i}]\!]_{\eta \oplus [x_1/d_1,\ldots,x_m/d_m]} & \text{falls } [\![\mathrm{e'}]\!]_\eta = (\underline{\mathrm{C_i}}, d_1, \ldots, d_m), \\ & \quad p_i = \underline{\mathrm{C_i}}\ x_1\ \ldots\ x_m \text{ und} \\ & \quad C(p_1), \ldots, C(p_{i-1}) \neq \underline{\mathrm{C_i}}, \\ \bot & \text{sonst.} \end{cases}$$

Beispiel: Sei $e = \text{case } e' \text{ of } \{\underline{\mathrm{Zero}} \mathrel{->} \underline{\mathrm{Zero}}\ ;\ \underline{\mathrm{Succ}}\ x \mathrel{->} x\}$. Im Fall, dass $[\![\mathrm{e'}]\!]_\eta = \bot$, ist auch $[\![\mathrm{e}]\!]_\eta = \bot$. Wenn $[\![\mathrm{e'}]\!]_\eta = \underline{\mathrm{True}}$, dann ist auch $[\![\mathrm{e}]\!]_\eta = \bot$, weil $\underline{\mathrm{True}}$ nicht auf $\underline{\mathrm{Zero}}$ oder $\underline{\mathrm{Succ}}$ passt. Wenn $[\![\mathrm{e'}]\!]_\eta = (\underline{\mathrm{Succ}}, \underline{\mathrm{Zero}})$, dann ist $[\![\mathrm{e}]\!]_\eta = [\![\mathrm{x}]\!]_{\eta \oplus [x/\underline{\mathrm{Zero}}]} = \underline{\mathrm{Zero}}$.

Semantik

Wir müssten eigentlich noch zeigen, dass alle Funktionen, die wir in der Definition der Semantik benutzt haben, auch stetig sind. Da dies allerdings sehr aufwändig ist, müssen wir aus zeitlichen Gründen leider darauf verzichten.

Für ein gesamtes Programm nehmen wir an, dass es die Form $\underline{\text{decl}} \ \ldots \ \underline{\text{decl}} \ \underline{\text{expr}}$ hat, also $x_1 = e_1 \ \ldots \ x_n = e_n \ e$. Dies können wir schreiben als let $x_1 = e_1$ in $x_2 = e_2$ in $\ldots$ let $x_n = e_n$ in e. Zum Beispiel können wir statt

```
f = \x -> \y -> x + y
g = \x -> f x 2
f 3
```

dann schreiben

```
let f = \x -> \y -> x + y
in let g = \x -> f x 2
in f 3
```

Typen

Abschließend wollen wir uns noch mit Typen beschäftigen. In unserer Semantik gibt es momentan viele Stellen, die zu „Laufzeitfehlern“ führen, welche man aber bei der Typüberprüfung schon ausschließen kann.

Zunächst wollen wir sicherstellen, dass bei $[\![\mathrm{e}]\!]_\eta$ jedes $x \in \mathrm{free}(e)$ auch einen Eintrag in η hat. Dazu müssen wir verlangen, dass jede Variable auch deklariert wurde.

Bei fundamentalen Funktionen wie $(+)$ gilt $[\![\mathrm{x} + \mathrm{y}]\!]_\eta = \bot$, wenn $[\![\mathrm{x}]\!]_\eta \notin \mathbb{Z}_\bot$ oder $[\![\mathrm{y}]\!]_\eta \notin \mathbb{Z}_\bot$, also zum Beispiel bei $[\![\mathrm{x}]\!]_\eta = \underline{\mathrm{True}}$. Dies können wir vermeiden, indem wir sicherstellen, dass nur Werte vom Typ Int an $(+)$ übergeben werden.

Typen

Bei der Applikation gilt $[\![(\mathrm{e}_1\mathrm{e}_2)]\!]_\eta = \bot$, wenn $[\![\mathrm{e}_1]\!]_\eta \notin [\mathrm{Dom} \to \mathrm{Dom}]$, also wenn $[\![\mathrm{e}_1]\!]_\eta$ keine Funktion ist. Wir müssen deshalb sicherstellen, dass e_1 einen Funktionstyp hat und dass e_2 den passenden Argumenttyp hat. Zum Beispiel sollte bei `f x y = x + y` die Funktion `f` den Typ `Int -> Int -> Int` bekommen. Ein Programm wie `f True 0` sollte also abgelehnt werden.

Bei case erhalten wir $\bot$, wenn die Patterns nicht die richtige Anzahl an Variablen haben. Außerdem wollen wir „unsinnige“ Patterns ausschließen. Beispiel:

```
f x = case x of { True -> 0 ; Nil -> False }
```

Hier kann `x` nicht sowohl den Typ `Bool` als auch den Typ `List t` für einen Typ `t` haben. Zweitens kann `f` nicht mehr zur Laufzeit entscheiden, ob es einen Wert vom Typ `Int` oder `Bool` liefert.

Typen

Sprachen, die beim Kompilieren eines Programms Typüberprüfung machen, nennt man *statisch typisierte* Sprachen. Die Idee dabei ist, dass viele unsinnige Programme direkt abgelehnt werden und man nicht mühsam nach den dadurch verursachten Fehlern zur Laufzeit suchen muss.
Wegen der Unentscheidbarkeit des Halteproblems können wir nicht genau die Programme ablehnen, die zu Laufzeitfehlern führen.
Unser Typsystem wird sicherstellen, dass alle Programme mit Laufzeitfehlern abgelehnt werden, aber auch andere. Dies ist nötig, damit Wohlgetyptheit entscheidbar ist. Das Programm

```
let e = case True of
    { True -> 0 ; False -> False }
in e + 2
```

werden wir ablehnen, obwohl es nicht zu einem Laufzeitfehler führt.

Typen

Die Typen, die wir zunächst erlauben, sind folgendermaßen definiert: Int ist ein Typ. Wenn t und u Typen sind, dann ist $t \to u$ ein Typ. Welche Typen es noch gibt, hängt von den Data-Deklarationen in einem Programm ab. Wir werden zunächst nur einfache Data-Deklarationen mit Kind $*$ betrachten, also zum Beispiel `Bool` und `IntPair`. Die Menge aller Typen bezeichnen wir mit Type.

Unser erstes Ziel ist es, formal zu definieren, welche Ausdrücke welche Typen bekommen dürfen. Dazu müssen wir ähnlich wie bei der Semantik in der Lage sein, Ausdrücken mit Variablen Typen zuzuordnen, also zum Beispiel $x + 0$. Wenn wir x mit Int belegen, dann hat $x + 0$ den Typ Int.
Eine *Typumgebung* ist eine partielle Funktion Var $\to$ Type. Wir benutzen hier dieselben Schreibweisen wie für normale Umgebungen, also $\oslash$ und $[x_1/\tau_1, \ldots, x_n/\tau_n]$.