# Funktionales Programmieren

## Teil 10

Carl Philipp Reh

Universität Siegen

24. November 2023

# Domains für rekursive Data-Deklarationen

Wir müssen nun noch rekursiven Data-Deklarationen und Data-Deklarationen mit Typparametern Domains zuordnen. Diese Konstruktionen sind mathematisch sehr aufwändig und können hier leider nur angedeutet werden.

Betrachten wir wieder

```
data Nat = Zero | Succ Nat
```

Dies führt zu der Gleichung

$$\mathcal{D}(\mathrm{Nat}) = \{\underline{\mathrm{Zero}}\}_\perp \oplus (\{\underline{\mathrm{Succ}}\} \times \mathcal{D}(\mathrm{Nat}))_\perp.$$

Ähnlich wie bei rekursiv definierten Funktionen lösen wir diese Gleichung, indem wir nach einem Fixpunkt suchen. Wir fangen mit $\{\perp\}$ für $\mathcal{D}(\mathrm{Nat})$ an und wenden dann wiederholt die folgende Operation an:

$$\lambda D.\{\underline{\mathrm{Zero}}\}_\perp \oplus (\{\underline{\mathrm{Succ}}\} \times D)_\perp$$

# Domains für rekursive Data-Deklarationen

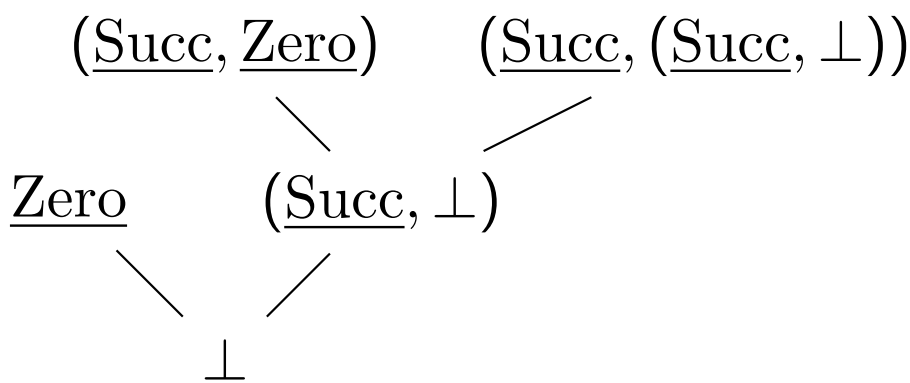
Wie bei $\mathcal{D}(\mathrm{Bool})$ lassen wir wieder die Tags weg. Im ersten Schritt erhalten wir also

$$\{\underline{\mathrm{Zero}}\}_\perp \oplus (\{\underline{\mathrm{Succ}}\} \times \{\perp\})_\perp = \{\perp, \underline{\mathrm{Zero}}, (\underline{\mathrm{Succ}}, \perp)\}$$

Setzen wir dies wieder ein, erhalten wir

$$\begin{aligned}&\{\underline{\mathrm{Zero}}\}_\perp \oplus (\{\underline{\mathrm{Succ}}\} \times \{\perp, \underline{\mathrm{Zero}}, (\underline{\mathrm{Succ}}, \perp)\})_\perp \\ &= \{\perp, \underline{\mathrm{Zero}}, (\underline{\mathrm{Succ}}, \perp), (\underline{\mathrm{Succ}}, \underline{\mathrm{Zero}}), (\underline{\mathrm{Succ}}, (\underline{\mathrm{Succ}}, \perp))\}\end{aligned}$$

Grafisch ergibt sich folgende Ordnung:



# Domains für rekursive Data-Deklarationen

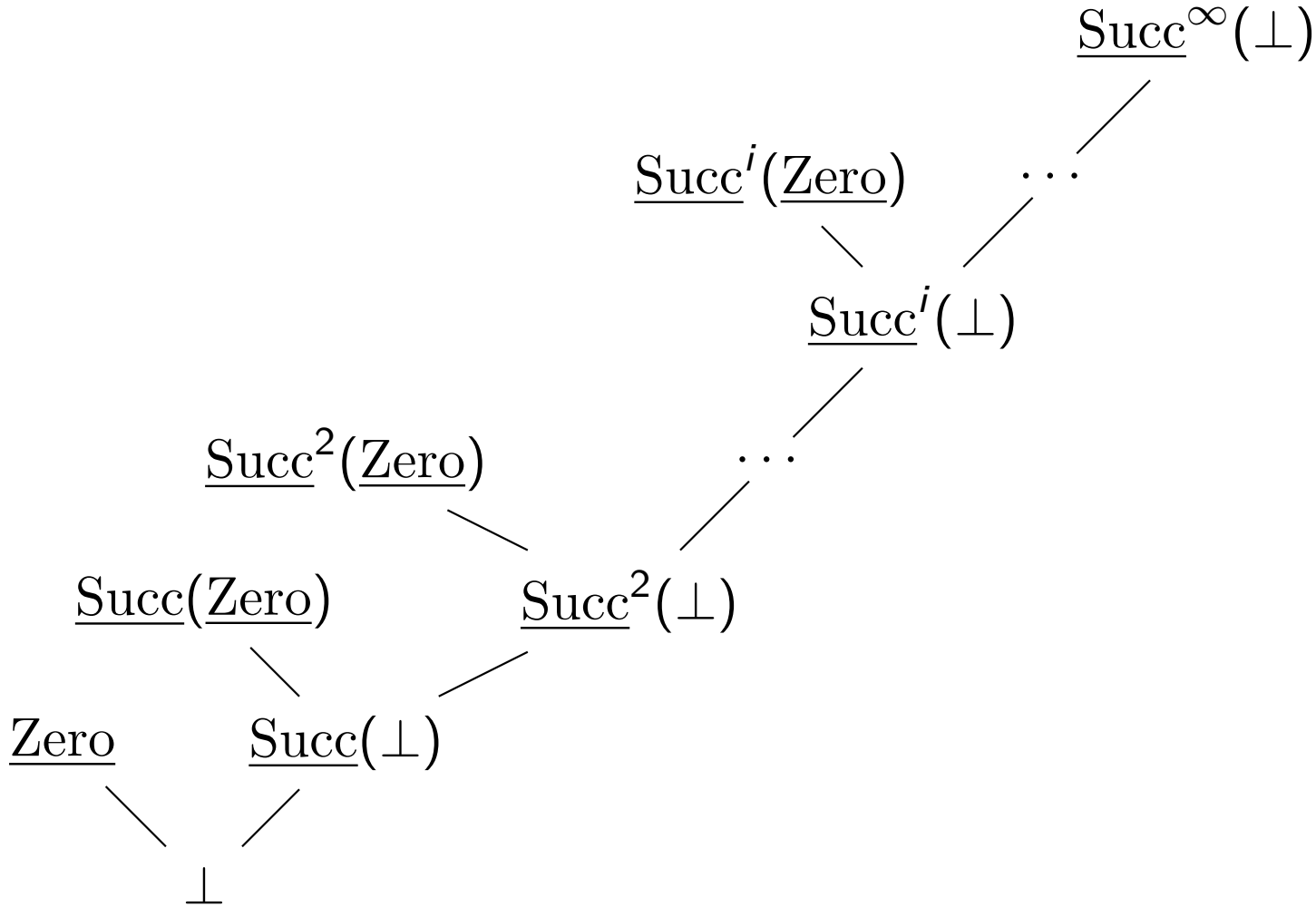
Wir schreiben nun $\underline{\text{Succ}}^n(x)$ statt $\underbrace{(\underline{\text{Succ}}, \cdots (\underline{\text{Succ}},}_{n \text{ mal } (\underline{\text{Succ}},} \underline{\text{x}} \underbrace{) \cdots )}_{n \text{ mal } )}$ für $n \in \mathbb{N}$. Jedes $\underline{\text{Succ}}^i(\underline{\text{Zero}})$ für $i \in \mathbb{N}$ kann man einfach über ein Haskell-Programm erzeugen, das $i$ mal `Succ` auf `Zero` anwendet. Ebenso kann man einfach $\underline{\text{Succ}}^i(\bot)$ erzeugen. Wir können allerdings noch einen weiteren Wert erzeugen. Betrachten wir:

```
inf :: Nat
inf = Succ inf
```

Die Semantik von `inf` kann nicht gleich $\underline{\text{Succ}}^i(\bot)$ für ein $i \in \mathbb{N}$ sein, weil man es $i + 1$ mal auf `Succ` matchen kann. Wir benötigen also noch einen weiteren Wert $\underline{\text{Succ}}^\infty(\bot)$, der für unendlich viele Anwendungen von $\underline{\text{Succ}}$ steht.

# Domains für rekursive Data-Deklarationen

Grafisch ergibt sich also folgender Domain:



# Domains für Data-Deklarationen mit Parametern

Betrachten wir als Beispiel wieder

```
data List a = Nil | Cons a (List a)
```

Für einen konkreten Typ `t` wählen wir dann $\mathcal{D}(\mathrm{List}(\mathrm{t}))$ als den kleinsten Fixpunkt von der Operation

$$\lambda D.\{\underline{\mathrm{Nil}}\}_\perp \oplus (\{\underline{\mathrm{Cons}}\} \times \mathcal{D}(\mathrm{t}) \times D)_\perp.$$

Ein Wert, der den Typ `List a` hat, wobei `a` eine Typvariable ist, hat auch den Typ `List t` für *alle* Typen `t`. Solch ein Wert ist zum Beispiel `Nil`. Wir wählen daher

$$\mathcal{D}(\mathrm{List}(\mathrm{a})) = \bigcap \{\mathcal{D}(\mathrm{List}(\mathrm{t})) \mid t \text{ ist ein Typ}\}.$$

Welche Typen es gibt, hängt allerdings davon ab, welche Data-Deklarationen in einem Programm vorkommen.

# Domain eines Programms

Zu $n \in \mathbb{N}$ sei $\mathrm{Con}_n$ die Menge aller $n$-stelligen Wertkonstruktoren, die in einem Programm vorkommen. Dann sei

$$\mathrm{Functions} = [\mathrm{Dom} \to \mathrm{Dom}]_\perp \text{ und}$$
$$\mathrm{Constructors}_n = (\mathrm{Con}_n \times \mathrm{Dom}^n)_\perp \text{ für } n \in \mathbb{N},$$

wobei wir für *den Domain* Dom *des Programms* die kleinste Menge nehmen, die folgende Gleichung erfüllt:

$$\mathrm{Dom} = \mathbb{Z}_\perp \oplus \mathrm{Functions} \oplus \mathrm{Constructors}_0 \oplus \mathrm{Constructors}_1 \oplus \cdots$$

Diese Menge enthält auch nicht typkorrekte Werte wie $(\underline{\mathrm{Succ}}, \underline{\mathrm{True}})$, die wegen Typüberprüfung zur Laufzeit nicht vorkommen können.
Einige Funktionen, die wir für die Semantik brauchen, werden der Einfachheit halber den Domain des Programms als Definitions- bzw. Zielbereich haben.

# Syntax von Haskell

Um formal die Semantik eines Haskell-Programms festzulegen, müssen wir zunächst definieren, welche Syntax wir erlauben. Wir werden hierbei einige Einschränkungen vornehmen, die allerdings keine wirklichen Einschränkungen sind, da man alle anderen Haskell-Programme in unsere erlaubte Syntax übersetzen kann.

Der Einfachheit nehmen wir an, dass ein Haskell-Programm aus Deklarationen folgender Form besteht:

$$\underline{\text{decl}} \to \underline{\text{var}} = \underline{\text{exp}}$$

Hier steht $\underline{\text{var}}$ für alle Variablennamen. Die Menge aller Variablen bezeichnen wir mit Var.

Beispiele für solche Deklarationen sind `x = 5` und `f = \x -> \y -> x + y`.

# Syntax von Haskell

Die erlaubten Ausdrücke, die wir mit Exp bezeichnen, sind durch folgende Grammatik gegeben:

$$
\begin{aligned}
\underline{\text{exp}} \rightarrow\ & \underline{\text{var}} \\
\mid\ & \underline{\text{constr}} \\
\mid\ & \underline{\text{integer}} \\
\mid\ & (\underline{\text{exp}}\ \underline{\text{exp}}) \\
\mid\ & \textbf{let}\ \underline{\text{var}} = \underline{\text{exp}}\ \textbf{in}\ \underline{\text{exp}} \\
\mid\ & \backslash\underline{\text{var}} -> \underline{\text{exp}} \\
\mid\ & \textbf{case}\ \underline{\text{exp}}\ \textbf{of}\ \{\underline{\text{pat}} -> \underline{\text{exp}}\ ;\ \ldots\ ;\ \underline{\text{pat}} -> \underline{\text{exp}}\}
\end{aligned}
$$

Hierbei erkennt $\underline{\text{constr}}$ alle Wertkonstruktornamen, also die Elemente aus allen $\text{Con}_n$ für $n \in \mathbb{N}$, und $\underline{\text{integer}}$ alle ganzen Zahlen. Die Grammatik für $\underline{\text{pat}}$ ist

$$\underline{\text{pat}} \rightarrow \underline{\text{constr}}\ \underline{\text{var}}\ \ldots\ \underline{\text{var}}$$

# Syntax von Haskell

($\underline{\text{exp}}$ $\underline{\text{exp}}$) ist die Applikation, also das Anwenden eines Parameters auf eine Funktion. Wir haben meistens die Klammern weggelassen, da vereinbart wird, dass man statt $(\cdots(e_1\ e_2)\cdots e_n)$ auch $e_1 \cdots e_n$ schreiben darf.
Operatoren behandeln wir wie folgt: Statt die Infixschreibweise wie `2 + 5` zu verwenden, muss man die Präfixschreibweise `(((+) 2) 5)` verwenden. Hierbei ist `(+)` eine bereits bekannte Variable mit vordefinierter Semantik.
Statt Case-Ausdrücke einzurücken, müssen wir {, } und ; verwenden. Zum Beispiel steht

```
case e of { [] -> 0 ; x : xs -> 1 }
```

für

```
case e of
  [] -> 0
  x : xs -> 1
```

# Umgebung

Wir nennen eine partielle Funktion $\eta\colon \text{Var} \to \text{Dom}$ eine *Umgebung*. Wir gehen davon aus, dass gewisse Operationen wie + in jeder Umgebung eingetragen sind, also zum Beispiel soll gelten, dass $\eta((+)) = f_+$, wobei $f_+\colon \text{Dom} \to \text{Dom} \to \text{Dom}$ mit

$$f_+(x)(y) = \begin{cases} x + y & \text{falls } x, y \in \mathbb{Z}, \\ \bot & \text{sonst.} \end{cases}$$

Die *Konkatenation* zweier Umgebungen $\eta_1$ und $\eta_2$ ist $(\eta_1 \oslash \eta_2)\colon \text{Var} \to \text{Dom}$ mit

$$(\eta_1 \oslash \eta_2)(x) = \begin{cases} \eta_2(x) & \text{falls } \eta_2(x) \text{ definiert ist,} \\ \eta_1(x) & \text{sonst.} \end{cases}$$

Das heißt, dass $\eta_2$ Einträge in $\eta_1$ überschreibt.

# Freie Variablen

Wir können einem Ausdruck nur eine Semantik zuordnen, wenn wir bereits eine Semantik für alle in ihm *frei vorkommenden Variablen* haben. Wir definieren dazu die Funktion free: $\mathsf{Exp} \to 2^{\mathsf{Var}}$ mit

$$
\begin{aligned}
\mathsf{free}(x) &= \{x\} \text{ für } x \in \mathsf{Var},\\
\mathsf{free}(\underline{\mathrm{C}}) &= \emptyset \text{ für } \underline{\mathrm{C}} \in \mathsf{Con}_n, n \in \mathbb{N},\\
\mathsf{free}(i) &= \emptyset \text{ für } i \in \mathbb{Z},\\
\mathsf{free}((e_1\ e_2)) &= \mathsf{free}(e_1) \cup \mathsf{free}(e_2),\\
\mathsf{free}(\textbf{let } x = e_1 \textbf{ in } e_2) &= (\mathsf{free}(e_1) \cup \mathsf{free}(e_2)) \setminus \{x\},\\
\mathsf{free}(\backslash x \mathrel{->} e) &= \mathsf{free}(e) \setminus \{x\}.
\end{aligned}
$$

Für $e = \textbf{case } e' \textbf{ of } \{p_1 \mathrel{->} e_1 \,;\, \cdots \,;\, p_n \mathrel{->} e_n\}$ nehmen wir

$$\mathsf{free}(e) = \mathsf{free}(e') \cup \bigcup \{\mathsf{free}(e_i) \setminus \mathsf{free}(p_i)) \mid 1 \leq i \leq n\}.$$

Die freien Variablen eines Patterns $p = \underline{\mathrm{C}}\ x_1 \ldots x_n$ sind $\mathsf{free}(p) = \{x_1, \ldots, x_n\}$.