Softwareparadigmen

Dieses Skriptum basiert auf der Softwareparadigmen Übung im Sommersemester 2011 und dem Vorlesungsskriptum 2007.
Vorlesung von Alexander Felfernig

Übungsskriptum verfasst von Daniel Gruß. Version 4. März 2012.
Fehlerfunde bitte melden an gruss@student.tugraz.at.

Inhaltsverzeichnis

Kapitel 1

Syntax

1.1 Grundlegende Definitionen

Definition 1.1 (Alphabet)**:** Ein Alphabet Σ ist eine endliche Menge von Symbolen.

Binärzahlen haben das Alphabet $\Sigma = \{0, 1\}$. Eine einfache Variante der Markup-Sprache HTML hat das Alphabet $\Sigma = \{<, /, >, \ldots, a, b, c, \ldots\}$.

Definition 1.2: Σ^* ist die Menge aller beliebigen Konkatenationen von Symbolen aus Σ. Ein Element aus Σ^* nennen wir Wort.

Für $\Sigma = \{0, 1\}$ ist $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$.

Definition 1.3 (Sprache)**:** Eine Sprache $\mathcal{L}$ ist eine Teilmenge ($\subseteq$) von Σ^*.

Sei $\Sigma = \{0, 1\}$. Definieren wir die Sprache der Binärzahlen $\mathcal{B}$, müssen wir zu jedem Wort aus Σ^* entscheiden ob dieses Wort in B enthalten ist. Im Fall der Binärzahlen könnten wir z.B. ε herausnehmen, dann ist die Sprache $\mathcal{B} = \Sigma^* \setminus \{\varepsilon\}$.

Würden wir die Programmiersprache $\mathcal{C}$ als formale Sprache definieren, so wäre ein gesamtes (gültiges) $\mathcal{C}$-Programm ein Wort der Sprache, also ein Element der Menge $\mathcal{C}$.

Definition 1.4 (Compiler)**:** Seien $\mathcal{A}$ und $\mathcal{B}$ Programmiersprachen. Ein Compiler ist ein Programm, welches Programme von $\mathcal{A}$ nach $\mathcal{B}$ übersetzt.

Ein einfacher Compiler würde beispielseweise Binärzahlen zu Dezimalzahlen übersetzen.

1.2 Grammatiken und Sprachen

Definition 1.5 (Grammatik)**:** Eine Grammatik ist ein 4-Tupel (V_N, V_T, S, Φ).

- V_N ist eine endliche Menge von Nonterminalen (entsprechen Zuständen der FSM),
- V_T ist eine endliche Menge von Terminalen (entsprechen Alphabet der Sprache),
- $S \in V_N$ das Startsymbol (wie Initialzustand der FSM),
- $\Phi = \{\alpha \to \beta\}$ eine endliche Menge von Produktionsregeln (Zustandsübergänge).
  α ist hierbei eine beliebige Aneinanderreihung von Terminalen und Nonterminalen, die zumindest ein Nonterminal enthält, also $(V_N \cup V_T)^* V_N (V_N \cup V_T)^*$.
  β ist eine beliebige Aneinanderreihung von Terminalen und Nonterminalen, einschließlich der leeren Menge, also $(V_N \cup V_T)^*$.

Um zu verifizieren wann ein Wort von einer Grammatik erzeugt werden kann müssen wir nun zuerst definieren was eine Ableitung ist.

Definition 1.6: Sei (V_N, V_T, S, Φ) eine Grammatik und $\alpha, \beta \in (V_N \cup V_T)^*$. Wenn es 2 Zeichenfolgen τ_1, τ_2 gibt, so dass $\alpha = \tau_1 A \tau_2$, $\beta = \tau_1 B \tau_2$ und $A \to B \in \Phi$, dann kann β direkt (in einem Schritt) von α abgeleitet werden ($\alpha \to \beta$).

Diese Definition ist nur geeignet um eine Aussage darüber zu treffen was in genau einem Schritt abgeleitet werden kann. Wir definieren daher die reflexive Hülle dieses Operators.

Definition 1.7: Sei (V_N, V_T, S, Φ) eine Grammatik und $\alpha, \beta \in (V_N \cup V_T)^*$. Wenn es $n \in \mathbb{N}$ Zeichenfolgen τ_1, τ_n gibt, so dass $\alpha \to \tau_1, \tau_1 \to \tau_2, \ldots, \tau_{n-1} \to \tau_n, \tau_n \to \beta$, dann kann β von α (in n Schritten) abgeleitet werden ($\alpha \xrightarrow{+} \beta$).

Diese Definition ist für $n \geq 1$ geeignet. Wenn $\alpha = \beta$ ist, wäre $n = 0$. Wir definieren die reflexive, transitive Hülle durch eine Verknüpfung dieser beiden Fälle.

Definition 1.8: Es gilt $\alpha \xrightarrow{*} \beta$, genau dann wenn $\alpha \xrightarrow{+} \beta$ oder $\alpha = \beta$ (reflexive, transitive Hülle).

Mit dieser Definition können wir alle Wörter ableiten die diese Grammatik produziert.

Definition 1.9: Sei (V_N, V_T, S, Φ) eine Grammatik G. G akzeptiert die Sprache $L(G) = \left\{ w \mid S \xrightarrow{*} w, w \in V_T^* \right\}$, d.h. die Menge aller Wörter w die in beliebig vielen Schritten aus dem Startsymbol S ableitbar sind und in der Menge aller beliebigen Konkatenationen von Terminalsymbolen V_T enthalten sind.

Die Äquivalenz von zwei Sprachen zu zeigen ist im Allgemeinen nicht trivial. Wenn wir versuchen eine Sprache $\mathcal{L}$ durch eine Grammatik G zu beschreiben ist es im Allgemeinen nicht trivial zu zeigen dass $L(G) = \mathcal{L}$.

An dieser Stelle möchten wir festzuhalten: Um die Gleichheit zweier Mengen (Sprachen) M, N zu zeigen muss gezeigt werden, dass jedes Element (Wort) aus M in N enthalten ist und jedes Element (Wort) aus N in M enthalten ist. Ungleichheit ist daher viel leichter zu zeigen, da es genügt ein Element (Wort) zu finden welches nicht in beiden Mengen (Sprachen) enthalten ist. Der geneigte Leser kann probieren die Gleichheit oder Ungleichheit der Sprache unserer oben definierten Grammatik und der Sprache der Binärzahlen

zu zeigen.

Definition 1.10: Ein Programm $P_{\mathcal{L}}$ welches für ein Wort w entscheidet ob es in der Sprache $\mathcal{L}$ enthalten ist (d.h. `true` dann und nur dann zurückliefert wenn es enthalten ist), nennen wir **Parser**.

1.3 Chomsky-Sprachhierarchie

Wir haben Produktionsregeln definiert durch $\alpha \to \beta$, wobei α zumindest ein Nonterminal enthält.

Definition 1.11: Eine Grammatik ist nach der Chomsky-Sprachhierarchie:

- allgemein/uneingeschränkt (unrestricted)
  Keine Restriktionen
- Kontext-sensitiv (context sensitive): $|\alpha| \leq |\beta|$
  Es werden nicht mehr Symbole gelöscht als produziert.
- Kontext-frei (context free): $|\alpha| \leq |\beta|, \quad \alpha \in V_N$
  Wie Kontext-sensitiv; außerdem muss α genau **ein** Non-Terminal sein
- regulär (regular): $|\alpha| \leq |\beta|, \quad \alpha \in V_N, \quad \beta = aA, \quad a \in V_T \cup \{\varepsilon\}, A \in V_N \cup \{\varepsilon\}$
  Wie Kontext-frei; außerdem ist $\beta = aA$ wobei a ein Terminal oder ε ist und A ein Nonterminal oder ε. (Anmerkung: $\varepsilon\varepsilon = \varepsilon$)

Es gilt $\mathbb{L}_{\text{regular}} \subset \mathbb{L}_{\text{context free}} \subset \mathbb{L}_{\text{context sensitive}} \subset \mathbb{L}_{\text{unrestricted}}$ ($\mathbb{L}_{\text{x}}$ Menge aller Sprachen der Stufe x).

Nicht alle Grammatiken können in eine äquivalente Grammatik einer stärker eingeschränkten Stufe umgewandelt werden. Um zu zeigen dass es sich um echte Teilmengen ($A \subset B$) handelt müssen wir zeigen dass alle Elemente aus A in B enthalten sind und mindestens ein Element aus B nicht in A enthalten ist.

Die Chomsky-Sprachhierarchie unterscheidet Sprachen anhand der Komplexität der produzierten Sprache.

1.4 Parser

Wir überspringen an dieser Stelle den BPARSE-Algorithmus (siehe Vorlesungsskriptum) und betrachten stattdessen einen Recursive Descent Parser (RDP).

Bei einem RDP werden alle Nonterminale in Funktionen übersetzt und diese Funktionen behandeln die verschiedenen Produktionsregeln. Die Eingabe wird in die Terminale unterteilt (auch Tokens genannt). Wir verwenden im Pseudo-Code die Variable `token`, die immer das aktuelle Token enthält, sowie die Funktion `nextToken()`, die `token` auf das nächste Token setzt. Der Parser ruft die Startfunktion auf und gibt `true` zurück wenn

`token` leer ist (d.h. das Ende der Eingabe erreicht wurde). `ERROR` führt dazu dass der Parser die Eingabe (das Wort) nicht akzeptiert.

Definition 1.12 (Mehrdeutig, Eindeutig)**:** Sei $G = (V_N, V_T, S, \Phi)$ eine Grammatik. Wenn es für ein Wort $w \in L(G)$ mehrere unterschiedliche Ableitungssequenzen ω, ψ, d.h. $S \to \omega_1, \omega_1 \to \ldots, \ldots \to \omega_n, \omega_n \to w$, $S \to \psi_1, \psi_1 \to \ldots, \ldots \to \psi_k, \omega_k \to w$ wobei $\exists \psi_i : \psi_i \neq \omega_i$, ist es mehrdeutig. Wenn eine Grammatik ist genau dann eindeutig, wenn sie nicht mehrdeutig ist.

Definition 1.13 (Linksrekursiv)**:** Eine Grammatik ist direkt linksrekursiv wenn sie eine Produktion der Form $A\alpha \to A\beta$ enthält, wobei A ein Nonterminal ist. Eine Grammatik ist indirekt linksrekursiv wenn sie Produktionen der Form $A\alpha \to A_1\beta_1, A_1\alpha_1 \to A_2\beta_2, \ldots, A_n\alpha_n \to A\beta_n$ enthält, wobei A, A_i Nonterminale sind. Eine Grammatik ist linksrekursiv wenn sie direkt oder indirekt linksrekursiv ist.

Nun können wir mit Sprachen und Grammatiken umgehen und diese nach ihrer Komplexität einstufen.

1.5 Lexikalische Analyse

Definition 1.14: Ein regulärer Ausdruck A ist wie folgt rekursiv definiert (mit regulären Ausdrücken Q und R):

$$A = \begin{cases} \varepsilon & \text{Leerstring} \\ t & \text{ein Terminal, d.h. } t \in V_T \\ Q|R & \text{entweder } A = Q \text{ oder } A = R \\ QR & \text{Konkatenation zweier regulärer Ausdrücke} \\ Q? & \text{entspricht } Q|\varepsilon, \text{ d.h. } Q \text{ ist optional} \\ Q* & \text{beliebige Konkatenation von } Q \text{ mit sich selbst, d.h. } A \in \{\varepsilon, Q, QQ, \ldots\} \\ Q+ & \text{entspricht } QQ*, \text{ d.h. mindestens ein } Q \\ (Q) & \text{Klammerung des Ausdrucks } Q \end{cases}$$

Zur Vereinfachung erlauben wir Variablen (Bezeichner) in regulären Ausdrücken:

Definition 1.15: Sei B die Menge aller Variablen (Bezeichnungen). Ein regulärer Ausdruck der eine Variable b aus B enthält ist ein erweiterter regulärer Ausdruck. Wenn E ein erweiterter regulärer Ausdruck ist und c eine Variable aus B, dann ist $c := E$ ist eine reguläre Definition.

Übersetzen wir dazu den erweiterten regulären Ausdruck zu einer Funktion so beobachten wir:

- Terminale werden zu `if`-Abfragen,

- `Q*` wird zu einem Schleifenkonstrukt,
- `Q|R` wird zu einer `if, else if, else`-Abfrage wobei der `else` Fall zu einem Ablehnen der Eingabe führt,
- `QR` bedeutet dass zuerst `Q` überprüft wird, danach `R`.

1.6 Grammatikalische Analyse

Bei kontext-freien oder regulären Grammatiken werden wir fortan nur noch die Produktionsregeln mit unterstrichenen Nonterminalen anschreiben, da die Grammatik dadurch vollständig definiert wird:

- V_N ist die Vereinigung über alle Symbole auf der linken Seite der Produktionsregeln (d.h. alle Nonterminale),
- V_T ist die Vereinigung aller unterstrichenen Zeichenfolgen (d.h. alle Terminale),
- S, das Startsymbol ist (soweit nicht anders festgehalten) das erste aufgeführte Nonterminal,
- Φ wird explizit angegeben.

1.7 LL(1)-Grammatiken

Definition 1.16 (LL(1)-Grammatik)**:** Eine kontextfreie Grammatik G ist eine LL(1)-Grammatik (ist in LL(1)-Form) wenn sie

- keine Linksrekursionen enthält
- keine Produktionen mit gleichen Prefixen für die selbe linke Seite enthält
- ermöglicht immer in einem Schritt (d.h. nur mit Kenntnis des nächsten Tokens) zu entscheiden, welche Produktionsregel zur Ableitung verwendet werden muss.

LL(1) steht für "**L**eft to right", "**L**eftmost derivation", "**1** Token Look-ahead".

Wir definieren einfache Regeln zwecks Auflösung von:

- **Indirekten Linksrekursionen**
  Gibt es Produktionen der Form $A \to \alpha B \beta$ sowie $B \to \gamma$, dann kann man die Produktion $A \to \alpha\gamma\beta$ einfügen. Wenn $\alpha = \varepsilon$ ist können so indirekte Linksrekursionen gefunden werden.
- **Direkten Linksrekursionen**
  Gibt es Produktionen der Form $A \to A\alpha$ und $A \to \beta$, dann kann man diese in Produktionen der Form $A \to \beta R$ sowie $R \to \alpha R|\varepsilon$ umwandeln.
- **Linksfaktorisierungen**
  Gibt es Produktionen der Form $A \to \alpha B$ sowie $A \to \alpha C$, wobei α der größte gemeinsame Prefix von αB und αC ist, so können wir diese Produktion durch $A \to \alpha R$ und $R \to B|C$ ersetzen. Der größte gemeinsame Prefix einer Sequenz

von Terminalen und Nonterminalen ist die größte gemeinsame Zeichenkette dieser Sequenzen. Der größte gemeinsame Prefix von if E then E else E und if E then E ist if E then E.

Mit diesen Regeln können wir oftmals Grammatiken in äquivalente LL(1)-Grammatiken umformen. Es gibt natürlich Grammatiken bei denen dies nicht möglich ist. In diesem Fall bleibt nur die Möglichkeit eine andere Parsing-Methode zu verwenden oder die Sprache zu verändern.

1.8 LL(1)-Tabellen

LL(1)-Tabellen (auch: LL(1)-Parser-Tabellen) ermöglichen es uns einen generischen LL(1)-Parser zu schreiben, der mit einer beliebigen Tabelle (und damit Sprache) arbeiten kann. Wir werden uns nun erarbeiten wie eine solche Tabelle berechnet werden kann. Für die Definition der FIRST- und Follow-Mengen stellen wir uns Nonterminale wieder als Zustände in einem Graph oder eine Maschine vor.

Definition 1.17 (FIRST-Menge)**:** Die FIRST-Menge eines Nonterminals X ist die Menge aller Terminalsymbole die im Zustand X **als erstes** geparst werden können. Die FIRST-Menge eines Terminals x ist immer das Terminalsymbol selbst.

Formal bedeutet dies:

1. Wenn x ein Terminal ist: $\text{FIRST}(x) = \{x\}$
2. Wenn die Grammatik Produktionsregeln enthält so dass $X \to \ldots \to \varepsilon$, dann ist: $\varepsilon \in \text{FIRST}(X)$
3. Für jede Produktionsregel $X \to Y_1 Y_2 \ldots Y_n$, ist $x \in \text{FIRST}(X)$ wenn $x \in \text{FIRST}(Y_i)$ und für alle Y_j mit $j < i$ gilt, dass $\varepsilon \in \text{FIRST}(Y_j)$.

Für aufeinanderfolgende Terminal- bzw. Nonterminalsymbole $X_1 X_2 \ldots X_n$:

1. Wenn $x \in \text{FIRST}(X_i)$ und für alle X_j mit $j < i$ gilt, dass $\varepsilon \in \text{FIRST}(X_j)$, dann ist $x \in \text{FIRST}(X_1 X_2 \ldots X_n)$.
2. Wenn für alle X_i $\varepsilon \in \text{FIRST}(X_i)$ ist, dann ist auch $\varepsilon \in \text{FIRST}(X_1 X_2 \ldots X_n)$.

Aus dieser Definition folgt beispielsweise für einfache Regeln $A \to B$, dass $\text{FIRST}(A) = (\text{FIRST}(B) \setminus \{\varepsilon\}) \cup \ldots$ ist.

Definition 1.18: Zwecks Übersichtlichkeit definieren wir die FIRST*-Menge als FIRST-Menge ohne ε:

$$\text{FIRST}^*(X) = \text{FIRST}(X) \setminus \{\varepsilon\}\,.$$

Definition 1.19: Jede Eingabe des Parsers endet mit dem "End of Input" Symbol \$.

Definition 1.20 (Follow-Menge)**:** Die Follow-Menge eines Nonterminals X ist die Menge aller Terminalsymbole die direkt auf Zustand X **folgen** können. Das heißt, alle Terminalsymbole die nach Abarbeitung des Zustands X als erstes geparst werden können.

Formal bedeutet dies:

1. Wenn X das Startsymbol ist, dann ist $\$ \in \text{FOLLOW}(X)$
2. Für alle Regeln der Form $A \to \alpha B \beta$ ist $\text{FIRST}^*(\beta) \subseteq \text{FOLLOW}(X)$
3. Für alle Regeln der Form $A \to \alpha B$ bzw. $A \to \alpha B \beta$ mit $\varepsilon \in \text{FIRST}(\beta)$ ist $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$

Algorithmus (Parse-Table)**:** Der Parse-Table Algorithmus berechnet aus einer Grammatik eine Parse-Table (auch LL(1)-Tabelle) M.

1. Für jede Produktion $X \to \alpha$:
   (a) Für jedes Element $y \in \text{FIRST}^*(\alpha)$ bzw. wenn $\alpha = y$:
      i. Füge $X \to \alpha$ in $M(X, y)$ ein.
   (b) Wenn $\varepsilon \in \text{FIRST}(\alpha)$:
      i. Für alle $y \in \text{FOLLOW}(X)$:
         A. Füge $X \to \alpha$ in $M(X, y)$ ein.

Leere Einträge in der Tabelle sind Fehlerfälle. Diese können auch explizit mit ERROR beschriftet werden.

Definition 1.21: Eine Grammatik ist eine LL(1)-Grammatik, wenn die berechnete Parse-Tabelle keine Mehrfacheinträge hat.

Definition 1.22: Eine LL(1)-Parsing Tabelle (oder auch Parsing-Tabelle) stellt einen vollständigen Parse-Vorgang dar. Jede Zeile entspricht einem Bearbeitungsschritt im Parse-Vorgang. In Spalten werden Stack, Eingabe und die angewandte Produktionsregel aufgetragen.

Algorithmus (LL(1)-Parsing mit Tabelle)**:** Sei X das oberste Stack-Element, t das aktuelle Token der Eingabe w und $\mathcal{L}$ die von der Grammatik akzeptierte Sprache.

1. Wenn X ein Non-Terminal ist:
   (a) Nimm den Wert von $M(X, t)$
   (b) Ist der Eintrag leer oder ein Fehlereintrag: Abbruch ($w \notin \mathcal{L}$).
   (c) Sonst: Ersetze das oberste Stack-Element X durch Produktion in umgekehrter Reihenfolge (WVU wenn $M(X, t) = X \to UVW$).
2. Andernfalls (X ist ein Terminal):
   (a) Wenn $X = t = \$$: Parsing erfolgreich ($w \notin \mathcal{L}$).
   (b) Sonst, wenn $X = t \neq \$$, dann nimm X vom Stack und gehe zum nächsten Token im Input.
   (c) Sonst: Abbruch ($w \notin \mathcal{L}$).

Diese Definition kann für andere Parser leicht angepasst werden.

Kapitel 2

Semantik von Programmiersprachen

In funktionalen Programmiersprachen besteht jedes Programm aus einer oder mehreren Funktionen.

Definition 2.1 (Funktion)**:** Eine Funktion ist eine Relation zwischen einer Menge A und einer Menge B. Jedem Element aus der Menge A wird genau ein Element der Menge B zugeordnet. Das heißt: Für jeden möglichen Eingabewert gibt es genau einen Ausgabewert.

2.1 Sprache $\mathcal{A}$ - einfache arithmetische Ausdrücke

Arithmetische Ausdrücke sind Funktionen. Wir können beispielsweise die Funktionen Addition, Subtraktion und Multiplikation von zwei Zahlen in $\mathbb{R}$ definieren mit einem Eingabewert in $\mathbb{R} \times \mathbb{R}$ und einen Ausgabewert in $\mathbb{R}$. Auch die Division können wir als Funktion definieren von $\mathbb{R} \times \mathbb{R} \setminus \{0\}$ (Division durch 0 schließen wir damit aus, da die Division in diesem Fall nicht als Funktion definiert ist) auf Ausgabewerte in $\mathbb{R}$.

Definition 2.2: Die Sprache $\mathcal{A}$ definieren wir mit Alphabet $\Sigma = \left\{\underline{0}, \ldots, \underline{9}, \underline{(}, \underline{)}, \underline{+}\right\}$. Zwecks Einfachheit definieren wir Ziffern (D, digits) und Zahlen:

- $\mathcal{A}_D = \{\underline{0}, \ldots, \underline{9}\}$
- ZAHL $= (\mathcal{A}_D \setminus \{\underline{0}\}\, \mathcal{A}_D^*) \cup \{\underline{0}\}$

Wir definieren die Sprache A nun nicht mehr über eine Grammatik sondern durch eine induktive Beschreibung (Basisfall und allgemeine Fälle):

1. ZAHL$\subset \mathcal{A}$
2. Wenn $x, y \in \mathcal{A}$, dann ist auch $\underline{(}x\underline{)}\underline{+}\underline{(}y\underline{)} \in \mathcal{A}$.

An dieser Stelle sei noch einmal darauf hingewiesen dass wir x, y nicht unterstreichen dürfen, da sie keine Sprachelemente sind sondern Platzhalter, mathematisch würde man sie auch als Variablen bezeichnen.

Definition 2.3: Die Semantik der Sprache $\mathcal{A}$ definieren wir durch:

1. $I_{\mathcal{A}}(x) = \langle x \rangle$ wenn $x \in \mathcal{A}_N$. x ist dabei eine Zeichenkette im Programm, $\langle x \rangle$ die entsprechende Repräsentation in $\mathbb{N}_0$.
2. $I_{\mathcal{A}}(\underline{(}x\underline{)} + \underline{(}y\underline{)}) = I_{\mathcal{A}}(x) + I_{\mathcal{A}}(y)$ wenn $x, y \in \mathcal{A}$.

2.2 Sprache $\mathcal{V}$ - arithmetische Ausdrücke mit Variablen

Wir erweitern die Sprache $\mathcal{A}$ durch Variablen und schaffen so eine mächtigere Sprache $\mathcal{V}$. Um mit Variablen umgehen zu können brauchen wir nun einerseits eine Menge zulässiger Variablennamen und andererseits eine Funktion die von Variablennamen auf eine Wertemenge der semantischen Ebene (z.B. $\mathbb{N}_0$) abbildet. Die Menge der zulässigen Variablennamen nennen wir IVS (Individuenvariablensymbole).

Definition 2.4: Zwecks Einfachheit erlauben wir nur wenige Variablennamen und definieren daher

$$\text{IVS} = \{\underline{a}, \underline{b}, \ldots, \underline{z}\} \cup \{\underline{x1}, \underline{x2}, \ldots\}\,.$$

Die Funktion die von Variablennamen auf eine Wertemenge abbildet nennen wir ω-Environment, (Variablen-)Umgebung. Man kann sich diese Funktion auch als Tabelle vorstellen bzw. in einem Interpreter als Tabelle implementieren.

Definition 2.5: Die Menge aller Environments sei

$$\text{ENV} = \bigcup_{x \in \text{IVS}, y \in \Lambda} \{(x, y)\}\,,$$

das heißt, die Vereinigung über alle Tupel Variablenname $x \in$ IVS und Wert auf semantischer Ebene $y \in \Lambda$.

Für die Sprache $\mathcal{V}$ ist $\Lambda = \mathbb{N}_0$.

Definition 2.6: Die Syntax der Sprache $\mathcal{V}$ ist definiert durch:

1. ZAHL$\subset \mathcal{V}$
2. IVS$\subset \mathcal{V}$
3. $\underline{(}x\underline{)}\underline{+}\underline{(}y\underline{)} \in \mathcal{V}$, wenn $x, y \in \mathcal{V}$.

Die Interpretation eines Programms hängt nun nicht mehr allein vom Programm selbst ab, sondern auch von den Werten der Variablen im ω-Environment.

Definition 2.7: Die Interpretationsfunktion $I_{\mathcal{V}} : \text{ENV} \times \mathcal{V} \to \Lambda$ weist jedem Tupel aus Environment und Programm einen Wert in Λ zu.

1. $I_{\mathcal{V}}(\omega, k) = \langle k \rangle$ wenn $k \in$ ZAHL, $\omega \in$ ENV.
2. $I_{\mathcal{V}}(\omega, v) = \omega(v)$ wenn $vk \in$ IVS, $\omega \in$ ENV.
3. $I_{\mathcal{V}}(\underline{(}x\underline{)} + \underline{(}y\underline{)}) = I_{\mathcal{V}}(\omega, x) + I_{\mathcal{V}}(\omega, y)$ wenn $x, y \in \mathcal{V}$, $\omega \in$ ENV.

2.3 Datentypen

Bisher haben wir eine Sprache nur für einen Datentypen definiert. Dieser war implizit in der Definition der Sprache drin (beispielsweise die natürlichen Zahlen). Derartige Definitionen erlauben kein Ersetzen des Datentyps ohne die Definition der Sprache wesentlich zu überarbeiten. Da wir dies aber häufig wollen werden wir nun zuerst Datentypen auf der semantischen Ebene und anschließend die Repräsentation von Datentypen auf der syntaktischen Ebene definieren.

Definition 2.8 (Datentyp)**:** Ein Datentyp ist ein Tupel $\Psi = (A, F, P, C)$ mit

- A: Grundmenge (Wertebereich)
- F: Menge von Funktionen $f_i : A^{k_i} \to A^{l_i}$.
  f_i ist die i-te Funktion in der Menge, k_i die Dimension vom Urbild (Dimension des Inputs, Anzahl der Funktionsargumente) und l_i die Dimension vom Bild der i-ten Funktion (Dimension des Outputs, Anzahl der Funktionsrückgabewerte).
- P: Menge von Prädikaten $p_i : A^{k_i} \to \{T, F\}$.
  p_i ist die i-te Funktion in der Menge und k_i die Dimension vom Urbild (Dimension des Inputs, Anzahl der Funktionsargumente) der i-ten Funktion.
- C: Menge von Konstanten c_i wobei $c \subseteq A$.

Die Mengen F^{Σ}, P^{Σ} und C^{Σ} enthalten die entsprechenden Symbole für die syntaktische Repräsentation:

- Funktionssymbole F^{Σ}: je ein Symbol f_i^{Σ} (z.B. Name der Funktion) für jede Funktion f_i
- Prädikatensymbole P^{Σ}: je ein Symbol p_i^{Σ} (z.B. Name des Prädikats) für jedes Prädikat p_i
- Konstantensymbol C^{Σ}: je ein Symbol c_i^{Σ} (z.B. ausgeschriebene Form der Konstante) für jede Konstante c_i

Konstanten sind eigentlich nur spezielle Funktionen (0 Argumente) und wir unterscheiden nur zwecks Übersicht. Wir können bei der Definition eines Datentyps oft (z.B. bei den verschiedenen Datentypen für Zahlen) auf bekannte algebraische Strukturen (Halbgruppen, etc.) zurückgreifen.

2.4 Sprache der Terme $\mathcal{T}$

Wir müssen nun um den Datentyp verwenden zu können eine grundlegende Sprache definieren die diesen Datentyp verwendet. Auf dieser Sprache können dann weitere Sprachen aufgebaut werden.

Definition 2.9 (Sprache der Terme $\mathcal{T}$)**:** Sei $\Psi = (A, F, P, C)$ ein Datentyp. Das Alphabet Σ ist dann eine Vereinigung aus den Mengen der

- IVS (Individuenvariablensymbole)
- Funktionssymbole F^Σ
- Prädikatensymbole P^Σ
- Konstantensymbol C^Σ
- $\underline{(}$,$\underline{)}$ und $\underline{,}$
- Sondersymbole (Keywords): $\underline{\text{if}}$, $\underline{\text{then}}$, $\underline{\text{else}}$, $\underline{\text{begin}}$, $\underline{\text{end}}$, ...

Die Syntax von $\mathcal{T} \subseteq \Sigma$ über einem beliebigen Datentypen ist dann definiert durch:

1. $C^\Sigma \subseteq \mathcal{T}$, d.h. Konstantensymbole sind Terme
2. IVS $\subseteq \mathcal{T}$, d.h. Individuenvariablensymbole sind Terme
3. Wenn f_i^Σ ein n-stelliges Funktionssymbol ist und $t_1, \ldots, t_n$ Terme, dann ist auch $f_i^\Sigma \underline{(} t_1 \underline{,} \ldots \underline{,} t_n \underline{)}$ ein Term (Unterstreichungen beachten!).

Die Semantik von $\mathcal{T}$ definieren wir durch die Interpretationsfunktion $I_\mathcal{T} : \text{ENV} \times \mathcal{T} \to A$.

1. $I_\mathcal{T}(\omega, c_i^\Sigma) = c_i$ mit $c_i \in C$ (Semantik-Ebene), $c_i^\Sigma \in C^\Sigma$ (Syntax-Ebene) und $\omega \in$ ENV.
2. $I_\mathcal{T}(\omega, v) = \omega(v)$ mit $v \in$ IVS und $\omega \in$ ENV.
3. $I_\mathcal{T}(\omega, f_i^\Sigma \underline{(} t_1 \underline{,} \ldots \underline{,} t_n \underline{)}) = f_i(I_\mathcal{T}(\omega, t_1), \ldots, I_\mathcal{T}(\omega, t_n))$ mit $f_i \in F$ (Semantik-Ebene), $f_i^\Sigma \in F^\Sigma$ (Syntax-Ebene) und $\omega \in$ ENV.

Laut Definition von $\mathcal{T}$ gibt es nur die Konstanten 0 und 1. Variablen können natürlich jeden beliebigen Wert in $\mathbb{Z}$ annehmen. Es kann aber auch gezeigt werden dass alle ganzen Zahlen durch einen variablenfreien Term dargestellt werden können. Daher ist es nicht nötig dass es für jede Zahl eine Konstante gibt. Terme sind rekursiv definiert. Die **vollständige Induktion** (ab hier auch Induktion genannt) ist die übliche Beweistechnik für Beweise über rekursive bzw. rekursiv definierte Ausdrücke.

Halten wir fest: Die **vollständige Induktion** besteht aus 3 Einzelschritten: In der **Induktionsbasis** werden ein oder mehrere Basisfälle direkt bewiesen. In der **Induktionshypothese** wird versucht eine allgemeine Aussage (eine Hypothese) zu treffen von der angenommen wird dass sie bis zum n-ten Fall gilt. Im **Induktionsschritt** gehen wir einen Schritt weiter, also in den Fall $n + 1$ und versuchen diesen zu beweisen. Hier muss unbedingt auf die Induktionshypothese zurückgegriffen werden, sonst wurde keine vollständige Induktion durchgeführt. Um eine Induktion durchführen zu können müssen die Elemente unbedingt aufzählbar sein (d.h. man muss eine eindeutige Reihenfolge/-

Sortierung für die Elemente angeben können).

2.5 Sprache der Konditionale COND ($\mathcal{C}$)

Wir hatten in der Sprache der Terme $\mathcal{T}$ noch nicht die Möglichkeit Prädikate zu nutzen obwohl Datentypen über Prädikate verfügen. Dazu definieren wir die Sprache der Konditionale **COND**. Wir schreiben in diesem Skriptum großteils nur $\mathcal{C}$ zwecks Übersichtlichkeit - $\mathcal{C}$ gesprochen "COND".

Definition 2.10 (Die Sprache COND ($\mathcal{C}$))**:** Die Syntax von $\mathcal{C}$ ist wie folgt definiert:

1. $\mathcal{T} \subseteq \mathcal{C}$, d.h. alle Terme sind Konditionale.
2. Wenn p_i^{Σ} ein n-stelliges Prädikatensymbol ist und $u_1, \ldots, u_n, t_1, t_2$ Konditionale, dann ist auch $\underline{\text{if}}\ p_i^{\Sigma}\underline{(}u_1\underline{,}\ldots\underline{,}u_n\underline{)}\ \underline{\text{then}}\ t_1\ \underline{\text{else}}\ t_2$ ein Konditional.

Die Semantikfunktion $I_{\mathcal{C}}$ definieren wir durch:

1. $I_{\mathcal{C}}(\omega, t) = I_{\mathcal{T}}(\omega, t)$, wenn $t \in \mathcal{T}$ und $\omega \in \text{ENV}$.
2. Für jedes Prädikat p_i gilt:
   - Wenn $p_i\left(I_{\mathcal{C}}(\omega, u_1), \ldots, I_{\mathcal{C}}(\omega, u_n)\right) = T$, dann gilt:

$$I_{\mathcal{C}}\left(\omega,\ \underline{\text{if}}\ p_i^{\Sigma}\underline{(}u_1\underline{,}\ldots\underline{,}u_n\underline{)}\ \underline{\text{then}}\ t_1\ \underline{\text{else}}\ t_2\right) = I_{\mathcal{C}}(\omega, t_1)$$

   - Sonst ist $p_i\left(I_{\mathcal{C}}(\omega, u_1), \ldots, I_{\mathcal{C}}(\omega, u_n)\right) = F$ und dann gilt:

$$I_{\mathcal{C}}\left(\omega,\ \underline{\text{if}}\ p_i^{\Sigma}\underline{(}u_1\underline{,}\ldots\underline{,}u_n\underline{)}\ \underline{\text{then}}\ t_1\ \underline{\text{else}}\ t_2\right) = I_{\mathcal{C}}(\omega, t_2)$$

Bei Verschachtelungen von Prädikaten in $\mathcal{C}$ ist zu beachten dass Prädikate direkt kein Teil der Sprache sind. Daher ist auch der Ausdruck if ist0?(ist1?(x)) then x else add0(x) nicht in $\mathcal{C}$! Allerdings ist if ist0?(if ist1?(x) then 110 else 010) then x else add0(x) in $\mathcal{C}$.

2.6 Rekursionen - Sprache der Ausdrücke EXP ($\mathcal{E}$)

Bisher war es nicht möglich Funktionen (insbesondere rekursive) zu definieren. Wir müssen das Konzept "Funktionsname" ähnlich wie die Variablennamen (IVS, Individuenvariablensymbole) zuerst einführen. Der Wert der einem Funktionsnamen zugeordnet wird ist dabei ein Programm. Wir definieren nun die Sprache **EXP** ($\mathcal{E}$). Wie schon bei **COND** kürzen wir auch hier **EXP** durch $\mathcal{E}$ ab - $\mathcal{E}$ gesprochen "EXP".

Definition 2.11: Die Menge der Funktionsvariablensymbole (FVS) enthält "Namen" aller Funktionen.

Definition 2.12: Die Menge der Funktionsenvironments bezeichnen wir mit FENV. Jedes Funktionsenvironment δ : FVS $\to \mathcal{E}$ liefert für ein Funktionsvariablensymbol die

Implementation in EXP ($\mathcal{E}$) zurück. $\delta\underline{X}$ bezeichne die Implementation der Funktion mit dem Namen $\underline{X}$.

Es muss unbedingt unterschieden werden zwischen

- Funktionen des Datentyps ($+$, $-$, etc.) sowie den dazugehörigen Funktionssymbolen (plus, minus, etc.)
- und "benutzerdefinierten Funktionen", d.h. Unterprogramme die auch in der Sprache implementiert werden. Diese bezeichnen wir als Funktionsvariablensymbole.

Definition 2.13 (Syntax von EXP ($\mathcal{E}$))**:** Die Syntax von $\mathcal{E}$ ist definiert wie folgt:

- $\mathcal{C} \subseteq \mathcal{E}$, d.h. alle Konditionale sind Ausdrücke (d.h. $\in \mathcal{E}$).
- Wenn f eine n-stellige Funktionenvariable ($\in$ FVS) ist und $t_1, \ldots, t_n$ sind Ausdrücke (d.h. $\in \mathcal{E}$), dann ist $f\underline{(}t_1\underline{,}\ldots\underline{,}t_n\underline{)}$ ein Ausdruck (d.h. $\in \mathcal{E}$).

Eine Funktion kann also über die Parameter hinaus keine weiteren Variablen haben. Beim Funktionsaufruf werden Parameter übergeben. Wir behandeln dabei zunächst nur call-by-value. Hierbei werden die als Parameter übergebenen Ausdrücke **vor** Ausführung der Funktion berechnet (interpretiert) und die Variablen der Funktion werden in einem neuen Variablenenvironment auf die entsprechenden Werte initialisiert.

Definition 2.14 (Semantik von EXP ($\mathcal{E}$))**:** Die Semantikfunktion $I_{\mathcal{E}}$ definieren wir durch:

1. $I_{\mathcal{E}}(\delta, \omega, c) = I_{\mathcal{C}}(\omega, c)$, wenn $c \in \mathcal{C}$ und $\omega \in$ ENV.
2. $I_{\mathcal{E}}\left(\delta, \omega, F\underline{(}t_1\underline{,}\ldots\underline{,}t_n\underline{)}\right)$ mit $F \in$ FENV. Funktionsaufruf mit call-by-value:
   (a) Definiere ω' als neues Environment mit $\omega'(x_i) = I_{\mathcal{E}}(\delta, \omega, t_i)$ (für $1 \leq i \leq n$) wobei x_i die Parameter der Funktion δF sind. Im Normalfall ist $x_i = \underline{xi}$, d.h. $x_1 = \underline{x1}$, $x_2 = \underline{x2}$, usw.
   (b) $I_{\mathcal{E}}\left(\delta, \omega, F\underline{(}t_1\underline{,}\ldots\underline{,}t_n\underline{)}\right) = I_{\mathcal{E}}(\delta, \omega', \delta F)$

Anmerkung: Mit den Definitionen 2.9, 2.10 und 2.14 halten wir zwecks Übersicht fest:

1. $I_{\mathcal{T}}(\omega, c_i^{\Sigma}) = c_i$ mit $c_i \in C$ (Semantik-Ebene), $c_i^{\Sigma} \in C^{\Sigma}$ (Syntax-Ebene) und $\omega \in$ ENV.
2. $I_{\mathcal{T}}(\omega, v) = \omega(v)$ mit $v \in$ IVS und $\omega \in$ ENV.
3. $I_{\mathcal{T}}(\omega, f_i^{\Sigma}\underline{(}t_1\underline{,}\ldots\underline{,}t_n\underline{)}) = f_i(I_{\mathcal{T}}(\omega, t_1), \ldots, I_{\mathcal{T}}(\omega, t_n))$ mit $f_i \in F$ (Semantik-Ebene), $f_i^{\Sigma} \in F^{\Sigma}$ (Syntax-Ebene) und $\omega \in$ ENV.
4. $I_{\mathcal{C}}(\omega, t) = I_{\mathcal{T}}(\omega, t)$, wenn $t \in \mathcal{T}$ und $\omega \in$ ENV.
5. Für jedes Prädikat p_i gilt:
   - Wenn $p_i(I_{\mathcal{C}}(\omega, u_1), \ldots, I_{\mathcal{C}}(\omega, u_n)) = T$, dann gilt:

$$I_{\mathcal{C}}\left(\omega, \underline{\text{if}}\ p_i^{\Sigma}\underline{(}u_1\underline{,}\ldots\underline{,}u_n\underline{)}\ \underline{\text{then}}\ t_1\ \underline{\text{else}}\ t_2\right) = I_{\mathcal{C}}(\omega, t_1)$$

- Sonst ist $p_i\left(I_{\mathcal{C}}\left(\omega, u_1\right), \ldots, I_{\mathcal{C}}\left(\omega, u_n\right)\right) = F$ und dann gilt:

$$I_{\mathcal{C}}\left(\omega, \underline{\text{if}}\ p_i^{\Sigma}\underline{(}u_1\underline{,}\ldots\underline{,}u_n\underline{)}\ \underline{\text{then}}\ t_1\ \underline{\text{else}}\ t_2\right) = I_{\mathcal{C}}\left(\omega, t_2\right)$$

6. $I_{\mathcal{E}}\left(\delta, \omega, F\underline{(}t_1\underline{,}\ldots\underline{,}t_n\underline{)}\right)$ mit $F \in \text{FENV}$. Funktionsaufruf mit call-by-value:
   (a) Definiere ω' als neues Environment mit $\omega'(x_i) = I_{\mathcal{E}}\left(\delta, \omega, t_i\right)$ (für $1 \leq i \leq n$) wobei x_i die Parameter der Funktion δF sind. Im Normalfall ist $x_i = \underline{xi}$, d.h. $x_1 = \underline{x}1$, $x_2 = \underline{x}2$, usw.
   (b) $I_{\mathcal{E}}\left(\delta, \omega, F\underline{(}t_1\underline{,}\ldots\underline{,}t_n\underline{)}\right) = I_{\mathcal{E}}\left(\delta, \omega', \delta F\right)$

Wir definieren eine Funktion $\underline{\text{func}}$ in $\mathcal{E}$ durch $\delta\underline{\text{func}} = \underline{\ldots}$. Im Rahmen der Übung ist es auch zulässig bei der Funktionsdefinition explizit andere Parameter anzugeben ($\delta\underline{\text{func(x,v)}} = \underline{\ldots}$) um die Parameternamen festzulegen. In unserem Fall wäre dann $x_1 = \underline{x}$ und $x_2 = \underline{v}$. Die Komplexität des $\mathcal{E}$-Interpreters wird dadurch nicht wesentlich beeinflusst.

Im Vorlesungsskriptum findet sich außerdem auf Seite 59 die Defintion für Funktionsaufrufe mittels Call-by-Name sowie obiges Beispiel für Call-by-Name durchgerechnet. Außerdem wird demonstriert, dass es Funktionen gibt (die in Teilbereichen undefiniert sind) die je nach Verfahren ein unterschiedliches Verhalten zeigen.

2.7 Datentyp der Listen $\mathcal{L}$

Der Datentyp der Listen kommt in vielen Sprachen vor. Wir verwenden dazu die Darstellung mit eckigen Klammern und groß geschriebenen Wörtern ([APE BEE CAT]).

Definition 2.15 (Atom)**:** Etwas ist ein Atom genau dann wenn kein [und kein] darin vorkommt.

Listenelemente die keine "echten" Listen sind (APE, BEE, CAT, etc.) werden Atom genannt. Alle anderen Listen sind keine Atome.

Definition 2.16 (Datentyp der Listen $\mathcal{L}$)**:**

- Grundmenge A:
  - ATOM $\subseteq A$, d.h. alle Atome sind Listen
  - $[] \subseteq A$, d.h. die leere Liste ist eine Liste
- Funktionen
  1. f_1 : first (liefert das erste Element einer Liste)
     - Wenn $a \in$ ATOM, dann ist first$(a) = []$.
     - first$([]) = []$.
     - Wenn $\forall i, \quad 1 \leq i \leq k : \quad \ell_i \in L$, dann ist first$([\ell_1 \ldots \ell_k]) = \ell_1$
  2. f_2 : rest (liefert die Liste ohne das erste Element)
     - Wenn $a \in$ ATOM, dann ist rest$(a) = []$.

  - rest([]) = [].
  - Wenn $\ell \in L$, dann ist rest([ℓ]) = []
  - Wenn $\forall i, \quad 1 \leq i \leq k : \quad \ell_i \in L$ und $k > 1$, dann ist rest($[\ell_1 \ell_2 \ldots \ell_k]$) = $[\ell_2 \ldots \ell_k]$

3. f_3 : build (nimmt 2 Listen entgegen und fügt die eine als erste Element in die andere ein)
  - Wenn $a \in$ ATOM und $\ell \in L$, dann ist build$(\ell, a) = a$.
  - Wenn $\ell \in L$, build$(\ell, []) = [\ell]$.
  - Wenn $\ell \in L$ und $\forall i, \quad 1 \leq i \leq k : \quad \ell_i \in L$, dann ist build$(\ell, [\ell_1 \ldots \ell_k]) = [\ell \ell_1 \ldots \ell_k]$

- Prädikate
  1. p_1 : atom?
     - atom?$(x) = T$, genau dann wenn $x \in$ ATOM
  2. p_2 : eq?
     - eq?$(x, y) = T$, genau dann wenn $x = y$
- Konstanten: Je eine Konstante für jedes Atom, nil für die leere Liste

Die Symbole auf der syntaktischen Ebene werden entsprechend der zuvor verwendeten Bezeichnungen gewählt (z.B. $p_1^\Sigma = \underline{\text{atom?}}$).

2.8 Kodierung von Datentypen

In nahezu jedem Programm wird mehr als ein Datentyp verwendet. Nach unseren Definitionen können wir kein Programm schreiben welches die Länge einer Liste zurückliefert, da die Länge einer Liste eine Zahl in $\mathbb{N}_0$ ist und keine Liste.

Eine einfache Lösung für dieses Problem ist es Datentypen zu kombinieren. Wir erzeugen also einen neuen Datentyp der beispielsweise die Kombination aus L und $\mathbb{N}_0$ ist, wir schreiben dann $L + \mathbb{N}_0$.

Definition 2.17: Die Kombination von 2 Datentypen D_1, D_2 ergibt einen neuen Datentyp $D_1 + D_2$. Die Mengen A, F, P, C sowie die Mengen der syntaktischen Repräsentationen $A^\Sigma, F^\Sigma, P^\Sigma, C^\Sigma, \ldots$ des neuen Datentyps entsprechen der Vereinigung der jeweiligen Mengen der Datentypen D_1 und D_2.

Funktionen und Prädikate sind weiterhin nur für die Argumente aus dem eigenen Datentyp definiert. Im Falle eines Aufrufs mit Parametern eines fremden Datentyps wird ein definierter konstanter Wert (ein Fehlerwert) zurückgeliefert. Dazu können auch neue Konstanten eingeführt werden, welche dann jedoch wieder in allen Funktionen und Prädikaten behandelt werden müssen. Im Fall von Prädikaten ist der Fehlerwert entweder T oder F.

Die Definition von Atomen beim Datentyp der Listen erlaubt es den Listen leicht zu kombinieren. Elemente des zweiten Datentyps sind Atome.

Eine Alternative zur Kombination von Datentypen ist die **Kodierung** eines Datentyps in einem anderen Datentyp. Dies findet auch in vielen alltäglichen Programmen Anwendung - genau genommen immer dann wenn nicht nur die eingebauten Datentypen einer Sprache verwendet werden.

Definition 2.18 (Mapping-Funktion π)**:** Gegeben ein Datentyp $D = (A, F, P, C)$ und ein Datentyp E mit der Grundmenge B.

- Die Funktion $\pi : A \to B$ weist jedem Wert aus A einen Wert in B zu, d.h. $\forall x \in A : \pi(x) \in B$.
- $\forall f_i(\ldots) \in F$ wird ein Funktionausdruck $\pi[f_i](\ldots)$ über dem Datentyp E konstruiert welcher die Funktion implementiert.
- $\forall p_i(\ldots) \in P$ wird ein Prädikatenausdruck $\pi[p_i](\ldots)$ über dem Datentyp E konstruiert welcher das Prädikat implementiert.
- $\forall c_i \in C$ wird ein Funktionsausdruck (mglw. eine Konstante als 0-parametrige Funktion) $\pi[c_i]$ über dem Datentyp E konstruiert welcher die Konstante darstellt.

Definition 2.19 (Kodierungseigenschaften)**:**

- $\forall x \in A, f_i \in F : \pi[f_i](\pi(x)) = \pi(f_i(x))$, d.h. für alle Werte x in A und alle Funktionen in F darf es keinen Einfluss auf das Ergebnis haben, ob der Wert x zuerst kodiert und dann die kodierte Funktion mit diesem berechnet wird oder zuerst der die Funktion mit dem Wert x berechnet und das Ergebnis kodiert wird.
- $\forall x \in A, p_i \in P : \pi[p_i](\pi(x)) \leftrightarrow \pi(p_i(x))$, gleich wie bei den Funktionen

In Abbildung 2.1 ist die Definition als kommutierendes Diagramm dargestellt.

Abb. 2.1: Kodierungseigenschaften

Definition 2.20 (Dekodierungs-Funktion ρ)**:** ρ ist die Umkehrfunktion zur Mapping-Funktion π. Es gilt $\rho(\pi(x)) = x$.

2.9 Kodierung von $\mathcal{E}$ in den Datentyp der Listen

Wir wollen nun die Interpretationsfunktion ($I_{\mathcal{E}}$: FENV × ENV × $\mathcal{E} \to A$ mit einem Datentyp A) in $\mathcal{E}$ implementieren. Wir nennen diesen Interpreter δIX(delta,omega,exp. Die verschiedenen Input-Mengen müssen wir in Listen kodieren. Dabei unterscheiden wir

die Kodierungs- und Dekodierungsfunktionen: π_{FENV}, π_{ENV}, $\pi_{\mathcal{E}}$, π_A, die jedem Element der angegebenen Menge eine Repräsentation im Datentyp der Listen zuweisen.

Für den Interpreter sollen die Kodierungseigenschaften gelten:

$$I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{IX(delta,omega,exp)}}\right) = \pi_A\left(I_{\mathcal{E}}\left(\rho_{\text{FENV}}(\omega(\underline{\text{delta}})), \rho_{ENV}(\omega(\underline{\text{delta}})), \underline{\rho_{\mathcal{E}}(\omega(\underline{\text{exp}}))}\right)\right)$$

Der Einfachheit halber behandeln wir nur den Fall $A = L$, dann ist in unserem Fall $\pi_A(x) = x$. Die Kodierung von ω kann beispielsweise durch eine Liste von 2-elementigen Listen geschehen wobei das 1. Element jeweils der Variablenname und das 2. Element jeweils der Variablenwert ist. Das heißt der Sachverhalt $\omega(\underline{\text{x1}}) = \ell_1, \omega(\underline{\text{x2}}) = \ell_2$ wird dargestellt durch $[[\text{X1}\ \ell_1]\,[(X2)\,\ell_2]]$. Analog kann die Kodierung von FENV geschehen wobei hier 2 Werte pro Variable gespeichert werden: Die Parameterliste und das Programm in Listekodierung. δ hat dann folgende Form:

$$\left[\quad [F_1\,[[x_1 \ldots x_n]\,\pi_{\mathcal{E}}(\text{exp})]] \quad \ldots \quad \left[F_n\,\left[[x_1 \ldots x_n]\,\pi_{\mathcal{E}}(\text{exp}')\right]\right] \quad\right].$$

Wir definieren ℓ als Funktion die jedes Symbol in ein entsprechendes Atom übersetzt. D.h. $\ell(\underline{\text{first}}) = \text{FIRST}$, $\ell(\underline{\text{variable}}) = \text{VARIABLE}$, $\ell(\underline{\text{myfunc}}) = \text{MYFUNC}$, etc.

Definition 2.21 (Mapping-Funktion $\pi_{\mathcal{E}}$)**:** Wir definieren $\pi_{\mathcal{E}}$ so dass jeweils das erste Element einer Liste definiert ob diese Liste eine Konstante, Variable, Konditional, etc. ist.

$$\begin{aligned}
\forall c \in C^{\Sigma}: \quad & \pi_{\mathcal{E}}(c) = [\text{CONST}\ \ell(c)] && \text{(Konstanten)}\\
\forall v \in \text{IVS}: \quad & \pi_{\mathcal{E}}(v) = [\text{IVS}\ \ell(v)] && \text{(Variablen)}\\
\forall f \in F^{\Sigma}: \quad & \pi_{\mathcal{E}}(f\underline{(}t_1\underline{,}\ldots\underline{,}t_n\underline{)})\\
& = [\text{APPLY}\ \ell(f)\pi_{\mathcal{E}}(t_1)\ldots\pi_{\mathcal{E}}(t_n)] && \text{(Datentyp-Funktionen)}\\
\forall p \in P^{\Sigma}: \quad & \pi_{\mathcal{E}}(\underline{\text{if}}\ p\ \underline{(}u_1\underline{,}\ldots\underline{,}u_n\underline{)}\ \underline{\text{then}}\ t_1\ \underline{\text{else}}\ t_2)\\
& = [\text{COND}\ \ell(p)\,[\pi_{\mathcal{E}}(u_1)\ldots\pi_{\mathcal{E}}(u_n)]\,\pi_{\mathcal{E}}(t_1)\pi_{\mathcal{E}}(t_2)] && \text{(Datentyp-Prädikate)}\\
\forall F \in \text{FENV}: \quad & \pi_{\mathcal{E}}(F\underline{(}t_1\underline{,}\ldots\underline{,}t_n\underline{)})\\
& = [\text{CALL}\ \ell(F)\pi_{\mathcal{E}}(t_1)\ldots\pi_{\mathcal{E}}(t_n)]\\
& && \text{(Funktionen des Funktionsenvironments)}
\end{aligned}$$

Diese Definition erlaubt uns jedes $\mathcal{E}$-Programm in einer Liste zu kodieren.

2.10 Ein $\mathcal{E}$-Interpreter in $\mathcal{E}$

Die Implementation des $\mathcal{E}$-Interpreters $\delta\underline{\text{IX(delta, omega, exp)}}$ in $\mathcal{E}$ erfolgt nun direkt entsprechend der Definition 2.14 (sowie den dort verwendeten Definitionen 2.9 und 2.10)

wobei wir die größeren Teile der Definition auf Unterfunktionen aufteilen:

δIX(delta, omega, exp)
= if eq?(first(exp), CONST) then nth(exp, 2) (z.B. [CONST NIL])
else if eq?(first(exp), IVS) then DoIVS(omega, exp)
else if eq?(first(exp), APPLY) then DoAppl(delta, omega, exp)
else if eq?(first(exp), COND) then DoCond(delta, omega, exp)
else DoCall(delta, omega, exp)

δDoIVS(omega, exp) (Aufruf z.B. mit [IVS X1])
= if eq?(nth(exp, 2), first(first(omega))) then nth(first(omega), 2)
else DoIVS(rest(omega), exp)

δDoApply(delta, omega, exp) (Aufruf z.B. mit [APPLY BUILD X1 X2])
= if eq?(nth(exp, 2), FIRST) then first(IX(delta, omega, nth(exp, 3)))
else if eq?(nth(exp, 2), REST) then rest(IX(delta, omega, nth(exp, 3)))
else build(IX(delta, omega, nth(exp, 3)), IX(delta, omega, nth(exp, 4)))

δCheck(pred, exp)
= if eq?(pred, ATOM?) then if atom?(first(exp)) then TRUE else FALSE
else if eq?(exp) then if eq?(first(exp), nth(exp, 2)) TRUE else FALSE

δDoCond(delta, omega, exp) (Aufruf z.B. mit [COND P [X1] X2 X3])
= if eq?(Check(nth(exp, 2), nth(exp, 3)), TRUE) then IX(delta, omega, nth(exp, 4))
else IX(delta, omega, nth(exp, 5))

δbuild2(x, y)
= build(x, build(y, nil))

δNewEnv(params, values)
= if eq?(params, nil) then nil
else build(build2(first(params), first(values)), NewEnv(rest(params), rest(values)))

δFindFVS(delta, name)
= if eq?(name, first(first(delta))) then nth(first(delta), 2)
else FindFVS(rest(delta), name)

δDoCall(delta, omega, exp) (Aufruf z.B. mit [CALL F X1 X2 X3])
= IX(delta, NewEnv(FindFVS(delta, first(nth(exp, 2))), rest(rest(exp))),
FindFVS(delta, rest(nth(exp, 2))))

2.11 Das Halteproblem

$\mathcal{E}$ ist turing-vollständig. Der Beweis dazu kann erbracht werden indem man einen Interpreter für (in einer definierten Art und Weise) kodierte Turingmaschinen in EXP schreibt.

Definition 2.22 (Church-Turing These)**:** Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der intuitiv berechenbaren Funktionen.

Sofern die Church-Turing These stimmt gilt ein Beweis für ein Problem in EXP auch für alle anderen turing-vollständigen Sprachen.

Definition 2.23 (Entscheidungsproblem)**:** Ein Prädikat P ist entscheidbar, wenn es ein Programm Π in $\mathcal{E}$ mit folgender Eigenschaft gibt:

- Wenn $P(x_1, \ldots, x_n) = T$: $I_{\mathcal{E}}\,(\delta, \omega, \underline{\Pi}) = T$ mit $\omega(\underline{\text{xi}}) = x_i$
- Wenn $P(x_1, \ldots, x_n) = F$: $I_{\mathcal{E}}\,(\delta, \omega, \underline{\Pi}) = F$ mit $\omega(\underline{\text{xi}}) = x_i$

D.h. in einer endlichen Anzahl von Rechenschritten wird eine Entscheidung getroffen.

Ein Programm hält für eine Eingabe wenn es mit dieser Eingabe ausgeführt in einer endlichen Anzahl von Rechenschritten terminiert.

Definition 2.24 (Halteproblem)**:** Gibt es ein Programm das entscheiden kann ob ein beliebiges Programm Π auf einer Eingabe x hält?

Satz 2.1: Das Halteproblem ist unentscheidbar.

Beweis: Wir beweisen dass das Halteproblem (in $\mathcal{E}$) unentscheidbar ist, indem wir zeigen dass es kein Programm in $\mathcal{E}$ geben kann, welches für beliebige Programme und Inputs entscheidet ob das Programm hält. Dazu nehmen wir an es wäre entscheidbar in $\mathcal{E}$ und führen dies dann auf einen Widerspruch zurück. Diese Beweistechnik nennt man "Beweis durch Widerspruch".

Ein $\mathcal{E}$-Programm kann ein Programm aus mehreren Funktionen bestehen. Jedes $\mathcal{E}$-Programm kann aber auch mit nur einer Funktion implementiert werden. Sei L_Π die Listenkodierung einer Funktion Π.

Annahme: Es gibt eine Funktion $H(L_\Pi, X) : L \times L \to L$ die TRUE zurückgibt falls L_Π mit Input X hält, und FALSE zurückgibt falls L_Π mit Input X nicht hält und NIL als Fehlerwert zurückgibt falls L_Π keine $|X|$-stellige Funktion kodiert.

Für den Widerspruch zu zeigen:

$$\nexists\delta : \underline{\text{H(x,y)=}\ldots} \in \delta, \forall\omega \in \text{ENV} : I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{H(x,y)}}\right) = H(\omega(\underline{\text{x}}), \omega(\underline{\text{y}})),$$

d.h. es existiert kein Funktionsenvironment indem $\underline{\text{H(x,y)}}$ definiert wurde, so dass für alle möglichen ω-Environments $I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{H(x,y)}}\right) = H(\omega(\underline{\text{x}}), \omega(\underline{\text{y}}))$ berechnet werden kann.

Um dies zu zeigen konstruieren wir eine Funktion $\delta\underline{\text{A}}$, die genau dann halten soll wenn sie nicht hält:

$$\delta\underline{\text{A(x)}} = \underline{\text{if eq?(H(x,x),TRUE) then A(x) else x}}$$

Mit $\omega(\underline{\text{x}}) = \pi(\delta\underline{A})$ berechnen wir nun $I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{A(x)}}\right)$:

$$I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{A(x)}}\right) = I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{if eq?(H(x,x),TRUE) then A(x) else x}}\right) = \bigstar$$

Fallunterscheidung nach $I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{H(x,x)}}\right) =$

TRUE (d.h. $I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{A(x)}}\right)$ hält): Laut Definition der Interpretationsfunktion wird dann der then-Zweig ausgeführt.

$$\bigstar = I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{A(x)}}\right) = I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{if eq?(H(x,x),TRUE) then A(x) else x}}\right) = \bigstar$$

und da $I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{H(x,x)}}\right)$ =TRUE

$$\bigstar = I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{A(x)}}\right) = \ldots$$

Wir sehen dass das Programm nicht hält. Dies ist ein Widerspruch.

FALSE , d.h. $I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{A(x)}}\right)$ hält nicht. Laut Definition der Interpretationsfunktion wird dann der else-Zweig ausgeführt.

$$\begin{aligned} &= I_{\mathcal{E}}\left(\delta, \omega, \underline{\text{x}}\right) \\ &= \omega(\underline{\text{x}}) \end{aligned}$$

Wir sehen dass das Programm hält. Dies ist ebenfalls ein Widerspruch.

Die Fallunterscheidung deckt alle Fälle ab. Wir erhalten in jedem Fall einen Widerspruch. Somit haben wir den Satz bewiesen. □

Definition 2.25 (Äquivalenzproblem)**:** Gibt es ein Programm das entscheiden kann ob 2 Programme die selbe Funktion berechnen?

Satz 2.2: Das Äquivalenzproblem ist unentscheidbar.

Der Beweis ist eine gute Übungsaufgabe.

2.12 Sprache der Prädikatenlogischen Ausdrücke PL ($\mathcal{P}$)

TODO

2.13 Assignmentsprachen / Sprache AL ($\mathcal{A}$)

TODO

2.14 Die Sprache LP ($\mathcal{L}$)

TODO

2.15 Beweise in LP ($\mathcal{L}$)

TODO