TPML 1.0
Benutzerhandbuch

Christoph Fehling, Marcell Fischbach, Benedikt Meurer
Fachbereich 12 - Informatik und Elektrotechnik
Universität Siegen

12. Oktober 2006

Zusammenfassung

Die Vorlesung ,,Theorie der Programmierung" gehört zu den Pflichtveranstaltungen für Studierende der Angewandten Informatik im Hauptstudium, und vermittelt die Grundlagen zum Verständnis von modernen Programmiersprachen. Dies umfaßt operationelle Sematik und Typsysteme für unterschiedliche Sprachen. Die behandelten Sprachen basieren in der Regel auf OCaml. Da es allerdings mitunter schwierig sein kann, Zusammenhänge und bestimmte Details eines Beweises zu verstehen ohne den konkreten Ablauf des Interpreters oder des Typecheckers einmal Schritt für Schritt nachvollzogen zu haben, wurde im Rahmen einer Projektgruppe das Lernwerkzeug TPML entwickelt. Es ermöglicht den Studierenden das in der Vorlesung Gelernte direkt anzuwenden und dabei die einzelnen Schritte exakt zu verfolgen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Systemvoraussetzungen

1.2 Installation

1.3 Erste Schritte

Kapitel 2

Die Komponenten

2.1 Überblick über die Oberfläche

2.2 Der Editor

2.3 Der Small Step Interpreter

2.4 Der Big Step Interpreter

2.5 Der Type Checker

Kapitel 3

Die Sprachen im Detail

Dieses Kapitel beschreibt die in TPML verfügbaren Sprachen im Detail. Dies beinhaltet sowohl die abstrakte Syntax der Sprachen als auch die operationelle Semantik und das Typsystem, also die Regeln für die big und small step Interpreter und den Type Checker. Dieses Kapitel ist jedoch nicht geeignet als Ersatz für den Besuch der Vorlesung oder Übung, ebensowenig sollte dieses Handbuch als vollständiges Skript mißverstanden werden[1].

Die Sprachen $\mathcal{L}_0$ bis $\mathcal{L}_4$ sind strikt hierarchisch aufgebaut, das heißt die Sprache $\mathcal{L}_{n+1}$ erweitert die Sprache $\mathcal{L}_n$ $(0 \leq n < 4)$, beinhaltet also alle Merkmale der Sprache $\mathcal{L}_n$. Die Sprachen entsprechen im wesentlichen den in der Vorlesung behandelten Sprachen. Kleinere Abweichungen sind jedoch möglich und stellenweise nicht vermeidbar. In der Übung werden, falls notwendig, diese Abweichungen herausgestellt und erläutert.

3.1 Die Sprache $\mathcal{L}_0$

Die Sprache $\mathcal{L}_0$ stellt die einfachste denkbare Programmiersprache dar und entspricht dem reinen ungetypten λ-Kalkül (engl.: *pure untyped λ-calculus*). Die abstrakte Syntax enthält lediglich drei Produktionen.

$$\begin{array}{rcl} e & ::= & id \\ & | & \lambda id.\, e \\ & | & e_1 e_2 \end{array}$$

[1] Es mag wiederum allerdings nützlich als Grundlage für die Prüfungsvorbereitung sein, da es eine vollständige Auflistung des Regelwerks darstellt, welches allerdings nicht hunderprozentig mit dem Vorlesungsinhalt übereinstimmt.

Ein gültiger Ausdruck ist also entweder ein Bezeichner, eine λ-Abstraktion oder eine Applikation. Die Menge $Val \subseteq Exp$ der *Werte* (engl.: *values*) v wird durch

$$\begin{array}{rcl} v & ::= & id \\ & | & \lambda id.\, e \end{array}$$

definiert.

3.1.1 Big step Semantik von $\mathcal{L}_0$

Ein *big step* ist eine Formel der Gestalt $e \Downarrow v$ mit $r \in Val$. Ein big step heißt gültig für $\mathcal{L}_0$, wenn er sich mit den Regeln

$$\text{(VAL)} \qquad v \Downarrow v$$

$$\text{(BETA-V)} \qquad \frac{e[v/id] \Downarrow v'}{(\lambda id.\, e)\, v \Downarrow v'}$$

$$\text{(APP)} \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1\, v_2 \Downarrow v}{e_1\, e_2 \Downarrow v}$$

herleiten lässt.

3.1.2 Small step Semantik von $\mathcal{L}_0$

Ein *small step* ist eine Formel der Gestalt $e \to e'$. Ein small step heißt gültig für $\mathcal{L}_0$, wenn er sich mit den Regeln

$$\text{(BETA-V)} \qquad (\lambda id.\, e)\, v \to e[v/id]$$

$$\text{(APP-LEFT)} \qquad \frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2}$$

$$\text{(APP-RIGHT)} \qquad \frac{e \to e'}{v\, e \to v\, e'}$$

herleiten lässt.

3.2 Die Sprache $\mathcal{L}_1$

Die Sprache $\mathcal{L}_1$ erweitert $\mathcal{L}_0$ um Konstanten, bedingte Ausführung, den Bindungsmechanismus **let**, Ausnahmen (engl.: *exceptions*) und ein einfaches Typsystem, entspricht damit also dem einfach getypten λ-Kalkül. Vorgegeben seien:

- eine Menge *Exn* von *Ausnahmen* **exn**

  $$Exn = \{divide\_by\_zero\}$$

- für jeden arithmetischen Operator *op* eine Funktion

  $$op^{\mathcal{I}} : Int \times Int \to Int \cup Exn$$

- für jeden Vergleichsoperator *op* eine Funktion

  $$op^{\mathcal{I}} : Int \times Int \to Bool$$

Die Menge *Type* der *Typen* (engl.: *types*) τ ist definiert durch:

$$\begin{array}{rcl} \tau & ::= & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{unit} \\ & \mid & \tau_1 \to \tau_2 \end{array}$$

Die abstrakte Syntax, definiert durch die Menge *Exp* der gültigen Ausdrücke, wird erweitert durch neue Produktionen

$$\begin{array}{rcl} e & ::= & c \\ & \mid & \lambda id : \tau.\, e \\ & \mid & \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 \\ & \mid & \mathbf{let}\ id : \tau = e_1\ \mathbf{in}\ e_2 \\ & \mid & \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \\ & \mid & e_1\ op\ e_2 \\ & \mid & e_1\ \&\&\ e_2 \\ & \mid & e_1 \parallel e_2 \end{array}$$

wobei die Menge *Const* der *Konstanten* (engl.: *constants*) c durch

$$\begin{array}{rcll} c & ::= & () & \text{unit-Element} \\ & \mid & b \in \{true, false\} & \text{boolescher Wert} \\ & \mid & n \in \mathbb{Z} & \text{ganze Zahl} \\ & \mid & op & \text{Operator} \end{array}$$

und die Menge *Op* der *Operatoren* (engl.: *operators*) *op* durch

$$
\begin{array}{rcll}
op & ::= & + \mid - \mid * \mid / \mid \text{mod} & \text{arithmetische Operatoren} \\
& \mid & < \mid > \mid \leq \mid \geq \mid = & \text{Vergleichsoperatoren} \\
& \mid & not & \text{Negation}
\end{array}
$$

definiert ist. Die Menge *Val* der Werte v wird um die Produktionen

$$
\begin{array}{rcl}
v & ::= & c \\
& \mid & op\, e_1 \\
& \mid & \lambda id : \tau.\, e
\end{array}
$$

erweitert. Für Zahlkonstanten existiert derzeit die Einschränkung, dass nur positive Ziffernfolgen vom Lexer akzeptiert werden. Negative Zahlen können aber bei Bedarf durch Subtraktion konstruiert werden ($0 - n$ für $n \in \mathbb{N}$).

Die Angabe eines Typs bei λ-Abstraktion und **let** ist also optional, und für den big und small step Interpreter werden die Typangaben einfach ignoriert. Der Typechecker bestimmt bei $\lambda id.\, e$ den Typ für id mittels Typinferenz, während bei **let** die Angabe des Typs lediglich als zusätzliche Sicherheit für den Programmierer dient.

3.2.1 Big step Semantik von $\mathcal{L}_1$

Ein big step heißt gültig für $\mathcal{L}_1$, wenn er sich mit den big step Regeln von $\mathcal{L}_0$, den Regeln

(AND-FALSE) $\dfrac{e_1 \Downarrow \mathit{false}}{e_1 \,\&\&\, e_2 \Downarrow \mathit{false}}$

(AND-TRUE) $\dfrac{e_1 \Downarrow \mathit{true} \quad e_2 \Downarrow v}{e_1 \,\&\&\, e_2 \Downarrow v}$

(COND-TRUE) $\dfrac{e_0 \Downarrow \mathit{true} \quad e_1 \Downarrow v}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v}$

(COND-FALSE) $\dfrac{e_0 \Downarrow \mathit{false} \quad e_2 \Downarrow v}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v}$

(LET) $\dfrac{e_1 \Downarrow v_1 \quad e_2[v_1/id] \Downarrow v_2}{\textbf{let } id = e_1 \textbf{ in } e_2 \Downarrow v_2}$

(NOT) $\mathit{not}\, b \Downarrow \neg b$

(OP) $\mathit{op}\, n_1\, n_2 \Downarrow \mathit{op}^{\mathcal{I}}(n_1, n_2)$

(OR-FALSE) $\dfrac{e_1 \Downarrow \mathit{false} \quad e_2 \Downarrow v}{e_1 \,||\, e_2 \Downarrow v}$

(OR-TRUE) $\dfrac{e_1 \Downarrow \mathit{true}}{e_1 \,||\, e_2 \Downarrow \mathit{true}}$

und mit den zugehörigen exception-Regeln herleiten lässt. Diese exception-Regeln erhält man aus den obigen Regeln indem man zu jeder Regel der Form

(R) $\dfrac{e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n}{e \Downarrow v}$

(d.h. zu jeder Regel mit n Prämissen) für jedes $1 \leq i \leq n$ die Regel

(R-EXN-i) $\dfrac{e_1 \Downarrow v_1 \quad \ldots \quad e_{i-1} \Downarrow v_{i-1} \quad e_i \Downarrow \textbf{exn}}{e \Downarrow \textbf{exn}}$

hinzufügt. Exception-Regeln müssen beim big step Interpreter nicht explizit ausgewählt werden, sondern werden automatisch eingesetzt, sobald eine Ausnahme weitergereicht werden muß.

3.2.2 Small step Semantik von $\mathcal{L}_1$

Ein small step heißt gültig für $\mathcal{L}_1$, wenn er sich mit den small step Regeln von $\mathcal{L}_0$, den Regeln

(AND-EVAL) $\dfrac{e_1 \to e_1'}{e_1 \,\&\&\, e_2 \to e_1' \,\&\&\, e_2}$

(AND-FALSE) $\mathit{false} \,\&\&\, e_2 \to \mathit{false}$

(AND-TRUE) $\mathit{true} \,\&\&\, e_2 \to e_2$

(NOT) $\mathit{not}\, b \to \neg b$

(OP) $\mathit{op}\, n_1\, n_2 \to \mathit{op}^{\mathcal{I}}(n_1, n_2)$

(COND-EVAL) $\dfrac{e_0 \to e_0'}{\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \to \textbf{if } e_0' \textbf{ then } e_1 \textbf{ else } e_2}$

(COND-TRUE) $\textbf{if } \mathit{true} \textbf{ then } e_1 \textbf{ else } e_2 \to e_1$

(COND-FALSE) $\textbf{if } \mathit{false} \textbf{ then } e_1 \textbf{ else } e_2 \to e_2$

(LET-EVAL) $\dfrac{e_1 \to e_1'}{\textbf{let } \mathit{id} = e_1 \textbf{ in } e_2 \to \textbf{let } \mathit{id} = e_1' \textbf{ in } e_2}$

(LET-EXEC) $\textbf{let } \mathit{id} = v \textbf{ in } e \to e[v/\mathit{id}]$

(OR-EVAL) $\dfrac{e_1 \to e_1'}{e_1 \parallel e_2 \to e_1' \parallel e_2}$

(OR-FALSE) $\mathit{false} \parallel e_2 \to e_2$

(OR-TRUE) $\mathit{true} \parallel e_2 \to \mathit{true}$

und mit den zugehörigen exception-Regeln herleiten lässt. Diese exceptions-Regeln erhält man - ähnlich wie bei der big step Seminatik - aus den obigen Regeln indem man für jede Regel der Form

(R) $\dfrac{e_1 \to e_1'}{e_2 \to e_2'}$

(d.h. zu jeder Regel mit Prämisse) die Regel

(R-EXN) $\dfrac{e_1 \to \textbf{exn}}{e_2 \to \textbf{exn}}$

hinzunimmt. Wie beim big step Interpreter gilt auch für den small step Interpreter, dass exception-Regeln nicht explizit angegeben werden müssen.

3.2.3 Typsystem von $\mathcal{L}_1$

$\mathcal{L}_1$ verfügt über ein einfaches Typsystem, benutzt aber wie alle folgenden Sprachen schon den Typinferenzalgorithmus, was anfangs vielleicht zu schwer verständlichen Fehlermeldungen führen kann. In diesem Fall sollte es in der Übung angesprochen werden.

Ein *Typurteil für Ausdrücke* ist von der Form $\Gamma \triangleright e :: \tau$, wobei $\Gamma : Id \hookrightarrow Type$ eine partielle Funktion mit endlichem Definitionsbereich ist, die bestimmten Bezeichnern einen Typ zuordnet. Γ wird als *Typumgebung* (engl.: *type environment*) bezeichnet. Ein Typurteil heißt gültig für $\mathcal{L}_1$, wenn es sich mit den Regeln

$$\text{(CONST)} \quad \frac{c :: \tau}{\Gamma \triangleright c :: \tau}$$

$$\text{(ID)} \quad \Gamma \triangleright id :: \tau \text{ falls } id \in dom(\Gamma) \text{ und } \Gamma(id) = \tau$$

$$\text{(APP)} \quad \frac{\Gamma \triangleright e_1 :: \tau \to \tau' \quad \Gamma \triangleright e_2 :: \tau}{\Gamma \triangleright e_1\, e_2 :: \tau'}$$

$$\text{(COND)} \quad \frac{\Gamma \triangleright e_0 :: \mathbf{bool} \quad \Gamma \triangleright e_1 :: \tau \quad \Gamma \triangleright e_2 :: \tau}{\Gamma \triangleright \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 :: \tau}$$

$$\text{(LET)} \quad \frac{\Gamma \triangleright e_1 :: \tau_1 \quad \Gamma[\tau_1/id] \triangleright e_2 :: \tau_2}{\Gamma \triangleright \mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 :: \tau_2}$$

$$\text{(ABSTR)} \quad \frac{\Gamma[\tau/id] \triangleright e :: \tau'}{\Gamma \triangleright \lambda id : \tau.\, e :: \tau \to \tau'}$$

$$\text{(AND)} \quad \frac{\Gamma \triangleright e_1 :: \mathbf{bool} \quad \Gamma \triangleright e_2 :: \mathbf{bool}}{\Gamma \triangleright e_1 \,\&\&\, e_2 :: \mathbf{bool}}$$

$$\text{(OR)} \quad \frac{\Gamma \triangleright e_1 :: \mathbf{bool} \quad \Gamma \triangleright e_2 :: \mathbf{bool}}{\Gamma \triangleright e_1 \,\|\, e_2 :: \mathbf{bool}}$$

herleiten lässt. Die Regel (CONST) besagt hierbei, dass c in der Typumgebung Γ den Typ τ hat, wenn c den Typ τ hat. Dies wird durch die Regeln

(UNIT) $() :: \mathbf{unit}$

(BOOL) $b :: \mathbf{bool}$

(INT) $n :: \mathbf{int}$

(NOT) $not :: \mathbf{bool} \to \mathbf{bool}$

(AOP) $op :: \mathbf{int} \to \mathbf{int} \to \mathbf{int}$ falls op arithmetischer Operator

(ROP) $op :: \mathbf{int} \to \mathbf{int} \to \mathbf{bool}$ falls op Vergleichsoperator

definiert. Beim Typechecker werden diese Regeln für Konstanten nicht angegeben, sondern lediglich die (CONST) Regel.

3.2.4 Syntaktischer Zucker

Die Sprache $\mathcal{L}_1$ und alle folgenden Sprachen enthalten syntaktischen Zucker um das Schreiben von Programmen zu vereinfachen. Der syntaktische Zucker lässt sich dann anschließend in Kernsyntax übersetzen oder direkt verarbeiten. Hierbei wurden aus Gründen der Übersichtlichkeit nicht immer abgeleitete Regeln für den syntaktischen Zucker eingeführt, sondern es wird implizit eine Konvertierung des betreffenden Teilausdrucks in Kernsyntax vorgenommen. Für $\mathcal{L}_1$ betrifft dies Ausdrücke, die Operatoren in Infixschreibweise enthalten, und die logischen Operatoren && und ||. Es gilt:

$$e_1\ op\ e_2 \quad \text{steht für} \quad (op\ e_1)\ e_2$$

Beim small step Interpreter würden also die Regeln (APP-LEFT) und (OP) angewendet. Beim Schreiben von Programmen ist darüber hinaus zu beachten, dass Operatoren, sofern sie nicht in Infixausdrücken auftauchen, immer geklammert werden müssen. Die Funktion, die 1 zu ihrem Parameter addiert wird also als

$$(+)\,1$$

geschrieben. Dies entspricht der OCaml-Konvention und ist notwendig, da der Parser sonst bei bestimmten Ausdrücken nicht entscheiden kann, ob es ein Infixausdruck ist, oder der Operator als Parameter in eine Funktion eingesetzt werden soll. Zum Beispiel lässt sich

$$x + y$$

interpretieren als Infixaddition von x und y oder als Anwendung der Funktion x auf die Parameter $+$ und y. Intuitiv würde ein Mensch ersteres vermuten, der Parser für die konkrete Syntax kann dies jedoch nicht entscheiden.

Die logischen &&- und ||-Verknüpfungen sind syntaktischer Zucker für die bedingte Ausführung

$$\begin{array}{rll} e_1 \,\&\&\, e_2 & \text{steht für} & \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } \mathit{false} \\ e_1 \,\|\, e_2 & \text{steht für} & \textbf{if } e_1 \textbf{ then } \mathit{true} \textbf{ else } e_2 \end{array}$$

und es existieren abgeleitete Interpreter- und Typregeln, da sonst der Umgang mit diesen Konstrukten zu aufwendig wäre.

Interessant zu beobachten ist, dass auch der *not*-Operator als syntaktischer Zucker aufgefasst werden könnte.

$$\mathit{not} \quad \text{steht für} \quad \lambda id : \textbf{bool}.\, \textbf{if } id \textbf{ then } \mathit{false} \textbf{ else } \mathit{true}$$

Dies sei aber nur am Rande erwähnt.

3.3 Die Sprache $\mathcal{L}_2$

Die Sprache $\mathcal{L}_2$ erweitert $\mathcal{L}_1$ um rekursive Ausdrücke[2] und syntaktischen Zucker, der es erlaubt Funktionen einfacher zu definieren, ähnlich zu OCaml.

Die Menge *Exp* der gültigen Ausdrücke wird um die Produktionen

$$\begin{array}{rcl} e & ::= & \textbf{rec } id : \tau.\ e \\ & | & \textbf{let } id(id_1 : \tau_1) \ldots (id_n : \tau_n) : \tau = e_1 \textbf{ in } e_2 \\ & | & \textbf{let rec } id(id_1 : \tau_1) \ldots (id_n : \tau_n) : \tau = e_1 \textbf{ in } e_2 \end{array}$$

erweitert, wobei sämtliche Typangaben optional sind. Bei **let** und **let rec** müssen die Bezeichner für die Parameter nur dann geklammert werden, wenn ein Typ für diesen Parameter angegeben wird.

3.3.1 Big step Semantik von $\mathcal{L}_2$

Ein big step heißt gültig für $\mathcal{L}_2$, wenn er sich mit den big step Regeln von $\mathcal{L}_1$ und der Regel

$$\text{(UNFOLD)} \quad \frac{e[\textbf{rec } id.\ e/id] \Downarrow v}{\textbf{rec } id.\ e \Downarrow v}$$

sowie der dazugehörigen exception-Regel herleiten lässt.

[2] Wohlgemerkt aber nicht um rekursive Typen. Das Typsystem schränkt die Sprache also stärker ein, als dies bei $\mathcal{L}_1$ der Fall ist.

3.3.2 Small step Semantik von $\mathcal{L}_2$

Ein small step heißt gültig für $\mathcal{L}_2$, wenn er sich mit den small step Regeln von $\mathcal{L}_1$ und der Regel

(UNFOLD) $\quad \textbf{rec}\ id.\ e \rightarrow e[\textbf{rec}\ id.\ e/id]$

herleiten lässt.

3.3.3 Typsystem von $\mathcal{L}_2$

Ein Typurteil heißt gültig für $\mathcal{L}_2$, wenn es sich mit den Typregeln von $\mathcal{L}_1$ und der Regel

$$\text{(REC)} \quad \frac{\Gamma[\tau/id] \rhd e :: \tau}{\Gamma \rhd \textbf{rec}\ id : \tau.\ e :: \tau}$$

herleiten lässt. Fehlt die Angabe des Typs τ bei **rec** wird der Typ für e durch den Typinferenzalgorithmus bestimmt.

3.3.4 Syntaktischer Zucker

Die Sprache $\mathcal{L}_2$ enthält weitere Abkürzungen zu den in $\mathcal{L}_1$ definierten. Die folgenden Abkürzungen stehen für die leichtere Definition von Funktionen zur Verfügung.

$$\textbf{let}\ id(id_1 : \tau_1) \ldots (id_n : \tau_n) : \tau = e_1\ \textbf{in}\ e_2$$

steht für

$$\textbf{let}\ id : \tau = \lambda id_1 : \tau_1.\ \ldots \lambda id_n : \tau_n.\, e_1\ \textbf{in}\ e_2$$

und

$$\textbf{let rec}\, id(id_1 : \tau_1) \ldots (id_n : \tau_n) : \tau = e_1\ \textbf{in}\ e_2$$

steht für

$$\textbf{let}\ id : \tau = \textbf{rec}\ id : \tau.\ \lambda id_1 : \tau_1.\ \ldots \lambda id_n : \tau_n.\, e_1\ \textbf{in}\ e_2.$$

Für diesen syntaktischen Zucker wurden keine abgeleiteten Regeln eingeführt, stattdessen wird implizit eine Übersetzung in Kernsyntax vorgenommen.

3.4 Die Sprache $\mathcal{L}_3$

Die Sprache $\mathcal{L}_3$ erweitert die Sprache $\mathcal{L}_2$ um ein polymorphes Typsystem, sowie Tupel und Listen, für die zusätzlich abkürzende Schreibweisen eingeführt wurden. Die Menge *Exp* der gültigen Ausdrücke wird hierfür um die Produktionen

$$\begin{array}{rcll} e & ::= & (e_1, \ldots, e_n) \qquad (n \geq 2) & n\text{-Tupel} \\ & | & [e_1; \ldots; e_n] & \text{Liste} \\ & | & e_1 :: e_2 & \text{Konkatenation} \\ & | & \lambda(id_1, \ldots, id_n) : \tau_1 * \ldots * \tau_n . e & \\ & | & \textbf{let } (id_1, \ldots, id_n) : \tau_1 * \ldots * \tau_n = e1 \textbf{ in } e2 & \end{array}$$

erweitert, wobei sämtliche Typangaben wieder optional sind. Die Mengen *Const* und *Val* werden durch die Produktionen

$$\begin{array}{rcll} c & ::= & \mathit{fst} \mid \mathit{snd} & \text{Paarprojektionen} \\ & | & \#n\_i \qquad (1 \leq i \leq n) & \text{Projektionen} \\ & | & \mathit{cons} \mid [\,] & \text{Listenkonstruktion} \\ & | & \mathit{hd} \mid \mathit{tl} \mid \mathit{is\_empty} & \text{Listenoperatoren} \end{array}$$

und

$$\begin{array}{rcl} v & ::= & (v_1, \ldots, v_n) \\ & | & [v_1, \ldots, v_n] \\ & | & \mathit{cons}\, v_1 \end{array}$$

erweitert. Für Listenoperationen wird eine weitere Ausnahme

$$\textbf{exn} \quad ::= \quad \mathit{empty\_list}$$

hinzugenommen.

Die Menge der monomorphen Typen *Type* wird erweitert durch Produktionen für Tupel- und Listentypen, sowie Typvariablen $\alpha \ldots \omega$.

$$\begin{array}{rcll} \tau & ::= & \tau_1 * \ldots * \tau_n \qquad (n \geq 2) \\ & | & \tau' \textbf{ list} \\ & | & \alpha \mid \ldots \mid \omega \end{array}$$

3.4.1 Big step Semantik von $\mathcal{L}_3$

Ein big step heißt gültig für $\mathcal{L}_3$, wenn er sich mit den big step Regeln von $\mathcal{L}_2$ und den Regeln

(TUPLE)	$\dfrac{e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n}{(e_1, \ldots, e_n) \Downarrow (v_1, \ldots, v_n)}$
(FST)	$fst\,(v_1, v_2) \Downarrow v_1$
(SND)	$snd\,(v_1, v_2) \Downarrow v_2$
(PROJ)	$\#n\_i\,(v_1, \ldots, v_n) \Downarrow v_i \qquad (1 \leq i \leq n)$
(LIST)	$\dfrac{e_1 \Downarrow v_1 \quad \ldots \quad e_n \Downarrow v_n}{[e_1; \ldots; e_n] \Downarrow [v_1; \ldots; v_n]}$
(CONS)	$(::)\,v'\,[v_1; \ldots; v_n] \Downarrow [v'; v_1; \ldots; v_n]$
(HD)	$hd\ (cons\ (v_1, v_2)) \Downarrow v_1$
(HD-EMPTY)	$hd\,[\,] \Downarrow empty\_list$
(TL)	$tl\ (cons\ (v_1, v_2)) \Downarrow v_2$
(TL-EMPTY)	$tl\,[\,] \Downarrow empty\_list$
(IS-EMPTY-FALSE)	$is\_empty\ (cons\,v) \Downarrow false$
(IS-EMPTY-TRUE)	$is\_empty\,[\,] \Downarrow true$

sowie den entsprechenden exception-Regeln herleiten lässt. Zu beachten ist, dass - vielleicht wider der Intuition - die (CONS) Regel[3] für den binären Konkatenationsoperator :: definiert ist, und nicht für den unären Listenkonstruktor *cons*. Dies wird jedoch leicht ersichtlich, wenn man sich vor Augen hält, dass $(cons\,v) \in Val$. Für Details zum Thema Listen sei hier auf den Inhalt der Vorlesung verwiesen.

3.4.2 Small step Semantik von $\mathcal{L}_3$

Die small step Regeln für $\mathcal{L}_3$ entsprechen im wesentlichen den big step Regeln. Ein small step heißt gültig für $\mathcal{L}_3$, wenn er sich mit den small step Regeln von $\mathcal{L}_2$ und den Regeln

[3]Der Leser wird sich hoffentlich erinnern, dass binäre Operatoren, die außerhalb eines Infixausdrucks verwendet werden, geklammert werden müssen. $v' :: [v_1, \ldots, v_n]$ ist also syntaktischer Zucker für $(::)\,v'\,[v_1; \ldots; v_n]$.

(TUPLE)	$\dfrac{e_i \to e_i'}{(e_1, \ldots, e_i, \ldots, e_n) \to (e_1, \ldots, e_i', \ldots, e_n)}$
(FST)	$fst\,(v_1, v_2) \to v_1$
(SND)	$snd\,(v_1, v_2) \to v_2$
(PROJ)	$\#n\_i\,(v_1, \ldots, v_n) \to v_i \qquad (1 \leq i \leq n)$
(LIST)	$\dfrac{e_i \to e_i'}{[e_1; \ldots; e_i; \ldots, e_n] \to [e_1; \ldots; e_i'; \ldots; e_n]}$
(CONS)	$(::)\,v'\,[v_1; \ldots; v_n] \to [v'; v_1; \ldots; v_n]$
(HD)	$hd\ (cons\ (v_1, v_2)) \to v_1$
(HD-EMPTY)	$hd\,[\,] \to empty\_list$
(TL)	$tl\ (cons\ (v_1, v_2)) \to v_2$
(TL-EMPTY)	$tl\,[\,] \to empty\_list$
(IS-EMPTY-FALSE)	$is\_empty\ (cons\,v) \to false$
(IS-EMPTY-TRUE)	$is\_empty\,[\,] \to true$

sowie den entsprechenden exception-Regeln herleiten lässt.

3.4.3 Typsystem von $\mathcal{L}_3$

Wie bereits erwähnt verfügt die Sprache $\mathcal{L}_3$ über ein polymorphes Typsystem, das heißt dass die Axiome für Konstanten jetzt von der Form $c :: \pi$ sind. Für die Konstanten der Sprachen $\mathcal{L}_1$ bis $\mathcal{L}_2$ sei π einfach der bisherige monomorphe Typ τ, für die mit $\mathcal{L}_3$ neu hinzukommenden Konstanten sei es

der polymorphe Typ, wie im Folgenden dargestellt.

$$\begin{array}{rcl}
[\,] & :: & \forall\alpha.\,\alpha\,\textbf{list} \\
\mathit{cons} & :: & \forall\alpha.\,\alpha * \alpha\,\textbf{list} \to \alpha\,\textbf{list} \\
\mathit{hd} & :: & \forall\alpha.\,\alpha\,\textbf{list} \to \alpha \\
\mathit{tl} & :: & \forall\alpha.\,\alpha\,\textbf{list} \to \alpha\,\textbf{list} \\
\mathit{is_empty} & :: & \forall\alpha.\,\alpha\,\textbf{list} \to \textbf{bool} \\
:: & :: & \forall\alpha.\,\alpha \to \alpha\,\textbf{list} \to \alpha\,\textbf{list} \\
\mathit{fst} & :: & \forall\alpha_1,\alpha_2.\,\alpha_1 * \alpha_2 \to \alpha_1 \\
\mathit{snd} & :: & \forall\alpha_1,\alpha_2.\,\alpha_1 * \alpha_2 \to \alpha_2 \\
\#n_i & :: & \forall\alpha_1,\ldots,\alpha_n.\,\alpha_1 * \ldots * \alpha_n \to \alpha_i \quad (1 \le i \le n)
\end{array}$$

Die Regel

$$\text{(P-CONST)}\ \frac{c :: \pi}{\Gamma \rhd c :: \tau}\ \text{falls } \tau \text{ Instanz von } \pi$$

ersetzt die bisherige Regel (CONST), wobei τ' eine *neue Instanz* des polymorphen Typs $\pi = \forall\alpha_1,\ldots,\alpha_n.\,\tau$ ist, wenn τ' von der Form $\tau[\alpha'_1/\alpha_1,\ldots,\alpha'_n/\alpha_n]$ ist, wobei $\alpha'_1,\ldots,\alpha'_n$ neue Typvariablen sind.

In die Typumgebung $\Gamma : \mathit{Id} \hookrightarrow \mathit{PType}$ werden nun polymorphe Typen eingetragen. Entsprechend ersetzt die Regel

$$\text{(P-ID)}\ \ \Gamma \rhd \mathit{id} :: \tau \text{ falls } \mathit{id} \in \mathit{dom}(\Gamma),\ \Gamma(\mathit{id}) = \pi \text{ und } \tau \text{ Instanz von } \pi$$

die bisherige Regel (ID). Die neue Regel

$$\text{(P-LET)}\ \frac{\Gamma \rhd e_1 :: \tau \quad \Gamma[\mathit{Closure}_\Gamma(\tau)/\mathit{id}] \rhd e_2 :: \tau'}{\Gamma \rhd \textbf{let}\ \mathit{id} = e_1\ \textbf{in}\ e_2 :: \tau'}$$

sorgt für das „polymorph machen" des Typs beim Eintragen in die Typumgebung, wobei der polymorphe Abschluß eines Typs τ in der Typumgebung Γ durch

$$\mathit{Closure}_\Gamma(\tau) = \forall\alpha_1,\ldots,\alpha_n.\,\tau \text{ mit } \{\alpha_1,\ldots,\alpha_n\} = \mathit{free}(\tau)\backslash\mathit{free}(\Gamma)$$

definiert ist. Die Regel (LET) wird beibehalten, da sie für $\mathcal{L}_4$, wo (P-LET) eingeschränkt wird, noch benötigt wird. Für $\mathcal{L}_3$ ist es also zulässig, im Falle von $\mathit{Closure}_\Gamma(\tau) = \tau$ sowohl (LET) als auch (P-LET) anzuwenden[4].

[4]Naürlich auch dann, wenn es für die Wohlgetyptheit nicht erforderlich ist, dass für *id* ein polymorpher Typ eingetragen wird.

Für Listen und Tupel werden die Regeln

$$\text{(LIST)} \quad \frac{\Gamma \rhd e_1 :: \tau \quad \ldots \quad \Gamma \rhd e_n :: \tau}{\Gamma \rhd [e_1; \ldots; e_n] :: \tau\,\mathbf{list}}$$

$$\text{(TUPLE)} \quad \frac{\Gamma \rhd e_1 :: \tau_1 \quad \ldots \quad \Gamma \rhd e_n :: \tau_n}{\Gamma \rhd (e_1, \ldots, e_n) :: \tau_1 * \ldots * \tau_n}$$

hinzugenommen.

3.4.4 Syntaktischer Zucker

$\mathcal{L}_3$ enthält wie gesehen syntaktischen Zucker für den leichteren Umgang mit Listen und Tupeln. Die Akürzungen werden wie im Folgenden dargestellt in die Kernsyntax übersetzt.

$[e_1; \ldots; e_n]$	steht für	$cons\,(e_1, \ldots\, cons\,(e_n, [\,]))$	$(n \leq 1)$
$e_1 :: e_2$	steht für	$cons\,(e_1, e_2)$	
fst	steht für	$\#2\_1$	
snd	steht für	$\#2\_2$	

Mehrfaches λ

$$\lambda(id_1, \ldots, id_n) : \tau_1 * \ldots * \tau_n.\, e$$

steht für

$$\lambda id : \tau_1 * \ldots * \tau_n.\, \mathbf{let}\ id_1 = (\#n\_1\, id)\ \mathbf{in} \ldots \mathbf{let}\ id_n = (\#n\_n\, id)\ \mathbf{in}\, e$$

und mehrfaches **let**

$$\mathbf{let}\ (id_1, \ldots, id_n) : \tau_1 * \ldots * \tau_n = e_1\ \mathbf{in}\ e_2$$

steht für

$$\mathbf{let}\ id : \tau_1 * \ldots * \tau_n = e_1\ \mathbf{in}\ \mathbf{let}\ id_1 = (\#n\_1\, id)\ \mathbf{in} \ldots \mathbf{let}\ id_n = (\#n\_n\, id)\ \mathbf{in}\, e_2$$

wobei $id_1, \ldots, id_n$ verschieden sind und id ein neuer Name, der nicht in e bzw. e_2 und unter $id_1, \ldots, id_n$ vorkommt.

3.5 Die Sprache $\mathcal{L}_4$

Die Sprache $\mathcal{L}_4$ erweitert schließlich $\mathcal{L}_3$ um imperative Konzepte, also Speicher, sequentielle Ausführung und Schleifen. Die Menge *Exp* der gültigen Ausdrücke wird hierzu um die Produktionen

$$\begin{array}{rcl} e & ::= & \textbf{if } e_1 \textbf{ then } e_2 \\ & | & \textbf{while } e_1 \textbf{ do } e_2 \\ & | & e_1 \,;\, e_2 \end{array}$$

und die Menge *Const* um die Produktionen

$$c \;::=\; \mathit{ref} \mid ! \mid :=$$

erweitert.

3.5.1 Typsystem von $\mathcal{L}_4$

Ein Typurteil heißt gültig für $\mathcal{L}_2$, wenn es sich mit den Typregeln von $\mathcal{L}_1$ und den Regeln

$$\text{(SEQ)} \quad \frac{\Gamma \rhd e_1 :: \tau_1 \quad \Gamma \rhd e_2 :: \tau_2}{\Gamma \rhd e_1 \,;\, e_2 :: \tau_2}$$

$$\text{(WHILE)} \quad \frac{\Gamma \rhd e_1 :: \textbf{bool} \quad \Gamma \rhd e_2 :: \tau}{\Gamma \rhd \textbf{while } e_1 \textbf{ do } e_2 :: \textbf{unit}}$$

$$\text{(COND-1)} \quad \frac{\Gamma \rhd e_0 :: \textbf{bool} \quad \Gamma \rhd e_1 :: \textbf{unit}}{\Gamma \rhd \textbf{if } e_0 \textbf{ then } e_1 :: \textbf{unit}}$$

herleiten lässt. Die neuen Konstanten werden über die Regel (P-CONST) behandelt und haben die folgenden polymorphen Typen.

$$\begin{array}{rcl} ! & :: & \forall\alpha.\, \alpha\,\textbf{ref} \to \alpha \\ \mathit{ref} & :: & \forall\alpha.\, \alpha \to \alpha\,\textbf{ref} \\ := & :: & \forall\alpha.\, \alpha\,\textbf{ref} \to \alpha \to \textbf{unit} \end{array}$$

Um die Typsicherheit zu gewährleisten, muss die Regel (P-LET) eingeschränkt werden, und zwar wird nun nur noch über Werte quantifiziert

$$\text{(P-LET)} \quad \frac{\Gamma \rhd v_1 :: \tau \quad \Gamma[\mathit{Closure}_\Gamma(\tau)/\mathit{id}] \rhd e_2 :: \tau'}{\Gamma \rhd \textbf{let } \mathit{id} = v_1 \textbf{ in } e_2 :: \tau'}$$

wobei $v_1 \in \mathit{Val}$. Das bedeutet also in der Folge, dass (P-LET) nur noch angewendet werden darf, wenn hinter dem Gleichheitszeichen ein Wert steht. Sonst muss (LET) angewendet werden. Dies entspricht dem OCaml Typsystem.

3.5.2 Syntaktischer Zucker

Die Sprache $\mathcal{L}_4$ beinhaltet drei Abkürzungen, die wie folgt in Kernsyntax übersetzt werden

$$\begin{array}{rll} e_1\,;\,e_2 & \text{steht für} & \textbf{let}\ id = e_1\ \textbf{in}\ e_2 \\ \textbf{if}\ e_1\ \textbf{then}\ e_2 & \text{steht für} & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ () \\ \textbf{while}\ e_1\ \textbf{do}\ e_2 & \text{steht für} & \textbf{rec}\ id : \mathit{unit}.\ \textbf{if}\ e_1\ \textbf{then}\ (e_2\,;\,id) \end{array}$$

wobei $id \notin \mathit{free}(e_1) \cup \mathit{free}(e_2)$.

3.6 Konkrete Syntax

Da es mitunter nicht mit jeder herkömmlichen Tastatur möglich ist λ zu tippen kann stattdessen das Schlüsselwort **lambda** benutzt werden. Typvariablen $\alpha, \beta, \ldots$ werden in der üblichen OCaml-Schreibweise `'a`, `'b`, $\ldots$ geschrieben, und für Funktionstypen wird der Pfeil $\rightarrow$ als `->` notiert.