TPML 1.0
Benutzerhandbuch

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

24. 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

TPML wurde vollständig in Java entwickelt, und benötigt zur Ausführung lediglich ein JRE oder JDK in der Version 5.0 oder neuer. Einige System, wie zum Beispiel Mac OS X Tiger, diverse Linux Distributionen und angepasste Windows OEM Installation, enthalten bereits Java 5.0. Ansonsten kann die neueste Version der Java Runtime Environment (optional kombiniert mit dem Java Development Kit) von der Sun Microsystems Webseite heruntergeladen werden.

`http://java.sun.com/`

Bei manueller Installation des JRE oder JDK unter Unix/Linux-Systemen muss darauf geachtet werden, dass das `java`-Binary anschließend im `$PATH` verfügbar ist. Wird das JDK zum Beispiel unter `/usr/local/jdk1.5.0` installiert, muss anschließend im Profil der Shell `/usr/local/jdk1.5.0/bin` zum `$PATH` hinzugefügt werden. Im Falle der `bash` wäre dies in der Datei `.bashrc` im Homeverzeichnis ein Eintrag

```
export PATH="/usr/local/jdk1.5.0/bin:$PATH"
```

und anschließend muss entweder die Datei mittels `source .bashrc` neugeladen oder die Shell neu gestartet werden.

Unter Windows übernimmt das Java Installationsprogramm die Aufgabe die entsprechenden Systemeinstellungen anzupassen. Es sind in der Regel nach der Installation keine weitere Anpassungen mehr notwendig.

1.2 Installation

Das Programm steht als ZIP- und TAR-Archiv auf der TPML-Webseite zum Download bereit. Hier findet sich auch immer die jeweils aktuellste Version des Benutzerhandbuchs.

`http://tinyurl.com/y6ov4u`

Die Installation ist so einfach wie möglich gehalten. Nach dem Download des ZIP- oder TAR-Archivs muss dieses lediglich entpackt werden. Unter Mac OS X erledigt dies der StuffIt Expander, unter Windows kann in neueren Versionen entweder die in den Windows Explorer integrierte Unterstützung für komprimierte Ordner oder ein externes Programm wie zum Beispiel WinZIP verwendet werden. Unter Unix/Linux kann das TAR-Archiv mittels dem `tar`-Kommando entpackt werden[1].

```
bzcat tpml-X.Y.Z.tar.bz2 | tar xf -
```

Anschließend wechselt man in das erzeugte Verzeichnis `tpml-X.Y.Z` und führt dort das Shellskript `tpml.sh` aus. Sofern eine geeignete JavaSE 5.0 Installation gefunden wurde, startet nun TPML, ansonsten gibt es eine Fehlermeldung, und es sollte überprüft werden, ob wirklich ein JRE oder JDK 5.0 installiert ist, und sichergestellt werden, dass das `java`-Binary im `$PATH` liegt. Beim ersten Start von `tpml.sh` wird das Programm im System registriert, das heisst im Menu erscheint ein Eintrag für TPML und Dateien können in standardkonformen Dateimanagern (zum Beispiel Thunar und Nautilus) per Doppelklick geöffnet werden.

Unter Windows genügt ein Doppelklick auf die Datei `tpml.exe`. Sollte keine JavaSE 5.0 Installation gefunden werden, oder ist Java nicht korrekt im System eingerichtet erscheint ein Fehlerdialog.

1.3 Erste Schritte

Nach dem Start des Programms können neue Dateien erstellt werden. Dateien sind immer mit einer Programmiersprache verknüpft, die durch die Dateiendung identifiziert wird[2]. In TPML wurden die Sprachen $\mathcal{L}_0$ bis $\mathcal{L}_4$

[1] Oder alternativ natürlich auch über grafische Tools wie Xarchiver, File Roller oder Ark.

[2] In gleicher Weise wie `.java` eine Java-Datei bezeichnet und `.c` eine C-Datei.

realisiert, die den wesentlichen in der Vorlesung behandelten Sprachen entsprechen[3], wobei die Namen und der Umfang einzelner Sprachen von denen in der Vorlesung abweichen kann. Die Sprachen werden in Kapitel 3 im Detail behandelt, welches auch als Nachschlagewerk dienen soll.

Um eine neue Datei zu erstellen wählt man aus dem Menu **Datei** den Eintrag **Neu...** worauf sich ein Dialog öffnet, in dem die für die Datei zu benutzende Programmiersprache ausgewählt werden muss. Da die Vorlesung mit dem ungetypten λ-Kalkül beginnt, wählen wir also auch hier ein einfaches Programmbeispiel in der Programmiersprache $\mathcal{L}_0$. Nach der Auswahl der Programmiersprache öffnet sich ein Texteditor, in den der Programmtext eingegeben werden kann. Hierzu wählen wir das Beispiel der Identitätsfunktion, angewandt auf sich selbst.

```
(lambda x.x) (lambda x.x)
```

Zugegebenermaßen nicht das beindruckendste Programm, aber für ein erstes Beispiel doch sehr gut geeignet. Wie man bei der Eingabe des obigen Programms merken wird, unterstützt der Editor Syntaxhighlighting und markiert automatische Fehler im Programmtext, wie man es in der Regel von integrierten Entwicklungsumgebungen gewohnt ist[4].

Anschließend kann man nun den big oder small step Interpreter starten und einen Programmablauf durchspielen, der bei dem obigen Programm verständlicherweise nicht sonderlich spektakulär ist. Für das Beispiel benutzen wir den small step Interpreter. Dazu wählt man aus dem Menü **Beweis** den Eintrag **Small Step**. Es öffnet sich nun ein weiterer mit dieser Datei verbundener Reiter **Small Step** mit dem aus der Vorlesung bekannten Aussehen einer small step Herleitung. Der Interpreter erwartet nun die Auswahl einer small step Regel um den Beweis zu vervollständigen. Hierzu klickt man den grauen Button, und wählt dann aus dem Menü die nächste anzuwendende Regel, in unserem Fall die (BETA-V) Regel.

Nach Anwendung der Regel zeigt der Interpreter den nächsten Beweisschritt, der in diesem Fall auch schon der letzte ist, da $\lambda x.x$ bereits ein Wert ist, der naturgemäß nicht weiter ausgewertet werden kann.

Dies soll dem Zweck der *ersten Schritte* genügen. Das nächste Kapitel beschreibt die Benutzeroberfläche und die Beweiswerkzeuge im Detail.

[3]Derzeit existiert noch keine Sprache, die objekt-orientierte Konzepte unterstützt, wie sie in diesem Semester im zweiten Teil der Vorlesung eingeführt werden. Dies ist Aufgabe der folgenden Projektgruppe.

[4]Nichtsdestotrotz sollte TPML nicht mit einer IDE verwechselt, da es strikt als Lernwerkzeug ausgelegt ist.

Kapitel 2

Benutzerinterface

2.1 Überblick

Das Menu **Datei** enthält Funktionen zur Dateiverwaltung, wie beispielsweise öffnen und Schliessen. Weiterhin kann unter **Zuletzt benutzt** schnell auf die aktuellsten Dateien zugegriffen werden. Unter **Bearbeiten** sind Editorfunktionen enthalten. Die einzelnen Komponenten werden in Kapitel 2.2 und 2.3 genauer betrachtet. Mit **Einstellungen...** kann die Schriftart und Größe global für alle Editorkomponenten gesetzt werden. Weiterhin stellt man hier die Einfärbung von Schlüsselworten im Quelltexteditor 2.2 ein. Das Menu **Beweis** enthält die einzelnen Beweiswerkzeuge und dient dazu zwischen dem **Beginner** und **Fortgeschrittener** Modus zu wechseln. Diese Modi beeinträchtigen die bei den Beweisschritten zur Verfügung stehenden Regeln.

2.2 Quelltexteditor

Jede geöffnete Datei wird in einem Tab angezeigt, der den Dateinamen angibt. Zwischen den Tabs kann mit **Strg + Bildhoch/Bildrunter** und **Alt + Rechts/Links** gewechselt werden.

Jeder Datei ist eine Sprache fest zugeordnet. Möchte man einen Ausdruck in einer anderen Sprache beweisen, so kann man hierfür einen neuen Editor öffnen und der Ausdruck kopieren. Alternativ kann auch die Dateiendung einer gespeicherten Datei geändert werden. Ist ein eigegebener Ausdruck, so wird dies links von der betroffenen Zeile durch ein Error-Icon markiert. Zusätzlich wird der fehlerhafte Teil rot unterstrichen. Bewegt man den Mauszeiger über den markierten Text oder das Error-Icon, so erscheint ein Popup,

das den Fehler genauer beschreibt. Ein Klick auf das Error-Icon bewirkt, dass der Textcursor an den fehlerhalfen Text springt.

2.3 Beweiswerkzeuge

Wird ein Beweiswerkzeug gestartet, so wird es zusätzlich zum Quelltexteditor im Tab der Datei angezeigt. Für jede Datei kann jeweils nur ein Small Step Interpreter, ein Big Step Interpreter und ein Type Checker aktiv sein. Diese können über die entsprechende Schaltflächen angewählt werden. Wird ein Beweiswerkzeug erneut gestartet, so ersetzt der neue Beweis ein ggf. schon aktives Beweiswerkzeug des selben Typs. Möchte man also zum Beispiel einen Small Step Beweis neu starten oder für einen leicht geänderten Ausdruck erneut ausführen, so kann dies über eine erneute Anwahl der Small Step Funktion im Menu **Beweis** geschehen. Dabei wird jedoch der aktive Small Step Interpreter durch den neuen ersetzt.

Jedes Beweiswerkzeug verfügt über einen **Beginner** und einen **Fortgeschrittener** Modus. Diese Modi unterscheiden sich in den Regeln, zum Beweis des Ausdrucks zur Verfügung stehen. Zur Zeit betrifft diese Einstellung jedoch nur den Small Step Interpreter

2.4 Tutorials

Im folgenden wird anhand von einigen Beispielen in die Funktionsweise der einzelnen Beweiswerkzeuge eingeführt. Zunächst öffnen wir über **Datei** → **Neu...** einen Editor. Wie gesagt ist jeder Editor fest mit einer Sprache verknüpft.

Wir wählen für unser Beispiel die Sprache $\mathcal{L}_1$ aus und bestätigen mit **Ok**. Es ist nun standardmäßig der Quelltexteditor aktiv. Wir geben einen Ausdruck ein, mit dem wir nun arbeiten wollen:

(lambda x.x * 3) 4

Mit ein wenig Vorstellungskraft erkennt man, dass dieser Ausdruck voraussichtlich das Produkt aus 3 und 4 berechnet.

2.4.1 Small Step Interpreter

Unsere mutige Behauptung wollen wir nun natürlich durch einen stichhaltigen Beweis untermauern. Wir starten also mittels **Beweis** → **Small Step**

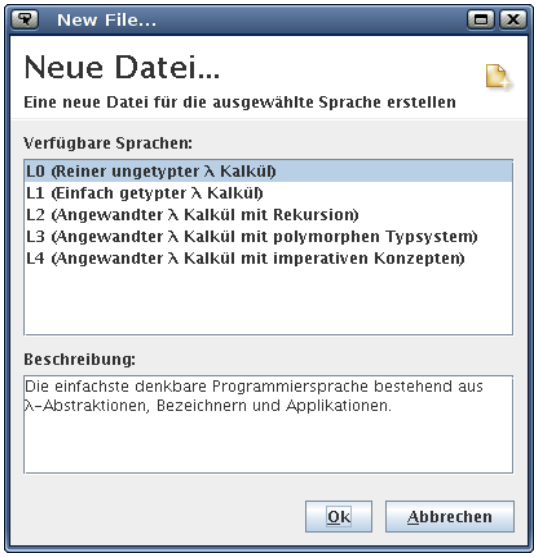

Abbildung 2.1: Eine neue Datei erstellen

oder mit der Taste F9 einen Small Step Interpreter.

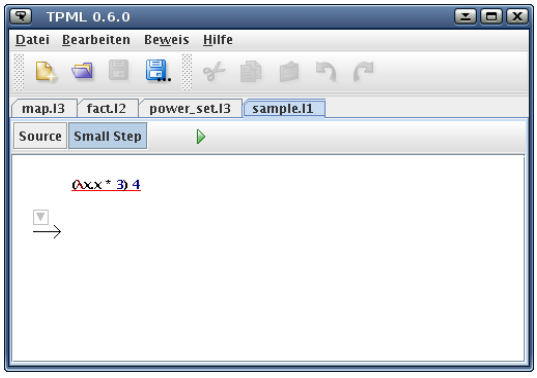

Abbildung 2.2: Der Small Step Interpreter

Bewegt man nun den Mauscursor über den Buttons des Regelmenus oberhalb des Pfeils für den ersten Ableitungsschritt wird der Teil des Ausdrucks rot unterstrichen, der als nächstes Abgeleitet werden soll, wie in Abbildung 2.2 gezeigt. In unserem Falle ist dies der gesamte Ausdruck. Der nächste Ableitungsschritt muss **BETA-V** sein, um die 4 für das x zu substituieren. Wir klicken also auf den Button und wählen in dem erscheinenden Regelmenu den entsprechenden Eintrag aus. Alternativ kann man auch einzelne Schritte oder den kompletten Beweis automatisch ausführen lassen. Für einzelne

Schritte klickt man entweder auf den grünen Pfeil oberhalb des Small Step Interpreters oder wählt **Raten** im Regelmenu. Eine komplette Beweisführung wird mittels **Vervollständigen** im selben Menu vorgenommen.

Ist man mit den Regeln vertraut, so kann man über **Beweis** → **Fortgeschrittener** einige Regeln ausblenden. Das Regelmenu enthält nun nur noch Regeln die spezifizieren was getan werden soll und nicht mehr genau wo. So muss zum Beispiel eine Bedingung nicht mehr mit **COND-EVAL** ausgewertet werden. Man gibt wählt lediglich **COND-TRUE** bzw. **COND-FALSE** aus.

2.4.2 Big Step Interpreter

Der Big Step Interpreter kann auch über das **Beweis** Menu oder mit der Taste F11 gestartet werden. Zu beachten ist, dass man sich hierfür nicht zwingend im Quelltexteditor befinden muss. Auch beim Big Step Interpreter können Regeln über das Regelmenu ausgewählt werden. Ist ein Unterbaum vollständig ausgewertet, so wird das ermittelte Ergebnis für höhere Knoten übernommen. Da unser Beispiel lediglich einen Unterbaum für die Regel **BETA-VALUE** enthält, ist dies auch gleichzeitig unser Ergebnis. Im Gegensatz zum Small Step Interpreter gibt es aufgrund der Baumstruktur des Beweises mehrere Stellen an denen Regeln angewandt werden können. Die Raten-Funktion bezieht sich dabei immer auf den obersten, noch nicht komplett ausgewerteten Teilbaum. Mit **Vervollständigen** wird stets nur der jeweilige Unterbaum komplett ausgewertet und nicht wie beim Small Step Interpreter der komplette Beweis zu ende geführt.

2.4.3 Type Checker

Diese Beweisart ist genau wie der Big Step Interpreter in einer Baumstruktur aufgebaut. Der Type Checker verhält sich daher bezüglich des Anwenden von Regeln, den Raten und der Vervollständigen Funktion genau wie der Big Step Interpreter. Das Anfügen von Typvariablen geschieht bei Regelanwendung automatisch.

Alternativ kann man mit der Funktion **Typ eingeben** in dem Regelmenu selbst einen Typ für den entsprechenden Teilbaum eingeben, wie in Abbildung 2.3 gezeigt. Der Typinferenzalgorithmus versucht dann den Unterbaum zu diesem Typ auszuwerten.

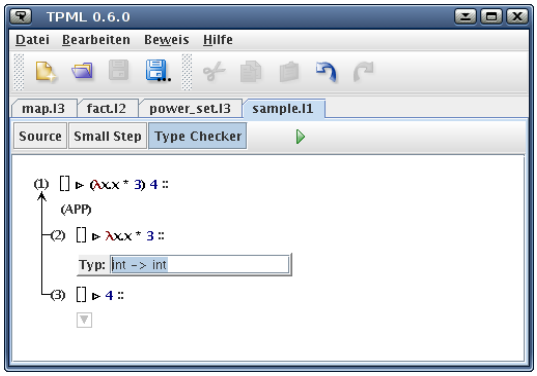

Abbildung 2.3: Der Type Checker

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 jedoch 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

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

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

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

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

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

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

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

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

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) $() :: \textbf{unit}$

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

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

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

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

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

definiert. Beim Type Checker 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

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

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

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

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

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

und

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

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 $\alpha \ldots \omega$.

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

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)	$hd\,[\,] \Downarrow \uparrow empty\_list$
(TL)	$tl\ (cons\ (v_1, v_2)) \Downarrow v_2$
(TL-EMPTY)	$tl\,[\,] \Downarrow \uparrow empty\_list$
(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)	$hd\,[\,] \to \uparrow empty\_list$
(TL)	$tl\ (cons\ (v_1, v_2)) \to v_2$
(TL-EMPTY)	$tl\,[\,] \to \uparrow empty\_list$
(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.

$$\begin{array}{rll} [e_1; \ldots; e_n] & \text{steht für} & \mathit{cons}\,(e_1, \ldots \mathit{cons}\,(e_n, [\,])) \quad (n \leq 1) \\ e_1 :: e_2 & \text{steht für} & \mathit{cons}\,(e_1, e_2) \\ \mathit{fst} & \text{steht für} & \#2\_1 \\ \mathit{snd} & \text{steht für} & \#2\_2 \end{array}$$

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.

Die operationelle Semantik muss für die Sprache $\mathcal{L}_4$ angepasst werden. Dazu sei eine unendliche Menge *Loc* vorgegen, deren Element $(X, Y, \ldots)$ *Speicherplätze* (engl.: *locations*)location genannt werden. Ein *Speicherzustand* (engl.: *store*) ist eine partielle Funktion

$$\sigma : \mathit{Loc} \hookrightarrow \mathit{Val}$$

mit endlichem Definitionsbereich $dom(\sigma)$. *Store* sei die Menge aller Speicherzustände. Eine *Konfiguration* ist entweder ein Paar $(e, \sigma) \in \mathit{Exp} \times \mathit{Store}$ oder $(exn, \sigma) \in \mathit{Exn} \times \mathit{Store}$. Für die Semantik des *ref*-Operators sei eine totale Funktion

$$\mathit{alloc} : \mathit{Store} \to \mathit{Loc}$$

mit $\mathit{alloc}(\sigma) \notin dom(\sigma)$ gegeben ($\mathit{alloc}(\sigma)$ ist also ein Speicherplatz in σ, der bisher noch nicht alloziert wurde).

3.5.1 Big step Semantik von $\mathcal{L}_4$

Ein big step ist von der Form $(e, \sigma) \Downarrow (v, \sigma')$ oder $(e, \sigma) \Downarrow (exn, \sigma')$. Ein big step heisst gültig für $\mathcal{L}_4$, wenn er sich mit den Regeln

(DEREF)	$(!\ X, \sigma) \Downarrow (\sigma(X), \sigma)$ falls $X \in dom(\sigma)$
(ASSIGN)	$(X\ :=\ v, \sigma) \Downarrow ((), \sigma[v/X])$ falls $X \in dom(\sigma)$
(REF)	$(\mathit{ref}\ v, \sigma) \Downarrow (X, \sigma[v/X])$ mit $X = \mathit{alloc}(\sigma)$
(SEQ)	$\dfrac{(e_1, \sigma) \Downarrow (v_1, \sigma') \quad (e_2, \sigma') \Downarrow (v_2, \sigma'')}{(e_1\,;\, e_2, \sigma) \Downarrow (v_2, \sigma'')}$
(COND-1-FALSE)	$\dfrac{(e_0, \sigma) \Downarrow (\mathit{false}, \sigma')}{(\textbf{if}\ e_0\ \textbf{then}\ e_1, \sigma) \Downarrow ((), \sigma')}$
(COND-1-TRUE)	$\dfrac{(e_0, \sigma) \Downarrow (\mathit{true}, \sigma') \quad (e_1, \sigma') \Downarrow (v_1, \sigma'')}{(\textbf{if}\ e_0\ \textbf{then}\ e_1, \sigma) \Downarrow (v_1, \sigma'')}$
(WHILE)	$\dfrac{(\textbf{if}\ e_1\ \textbf{then}\ e_2\,;\, \textbf{while}\ e_1\ \textbf{do}\ e_2, \sigma) \Downarrow (v, \sigma')}{(\textbf{while}\ e_1\ \textbf{do}\ e_2, \sigma) \Downarrow (v, \sigma')}$

sowie den entsprechenden exception-Regeln und den durch Speicherzustände erweiterten big step Regeln von $\mathcal{L}_3$ herleiten lässt.

3.5.2 Small step Semantik von $\mathcal{L}_4$

Ein small step ist nun von der Form $(e, \sigma) \to (e', \sigma')$ oder $(e, \sigma) \to (\mathit{exn}, \sigma)$. Ein small step heisst gültig für $\mathcal{L}_4$, wenn er sich mit den Regeln

(DEREF)	$(!\ X, \sigma) \to (\sigma(X), \sigma)$ falls $X \in dom(\sigma)$
(ASSIGN)	$(X\ :=\ v, \sigma) \to ((), \sigma[v/X])$ falls $X \in dom(\sigma)$
(REF)	$(\mathit{ref}\ v, \sigma) \to (X, \sigma[v/X])$ mit $X = \mathit{alloc}(\sigma)$
(SEQ-EVAL)	$\dfrac{(e_1, \sigma) \to (e_1', \sigma')}{(e_1\,;\, e_2, \sigma) \to (e_1'\,;\, e_2, \sigma)}$
(SEQ-EXEC)	$(v\,;\, e, \sigma) \to (e, \sigma)$
(COND-1-EVAL)	$\dfrac{(e_0, \sigma) \to (e_0', \sigma')}{(\textbf{if}\ e_0\ \textbf{then}\ e_1, \sigma) \to (\textbf{if}\ e_0'\ \textbf{then}\ e_1, \sigma)}$
(COND-1-FALSE)	$(\textbf{if}\ \mathit{false}\ \textbf{then}\ e, \sigma) \to ((), \sigma)$
(COND-1-TRUE)	$(\textbf{if}\ \mathit{true}\ \textbf{then}\ e, \sigma) \to (e, \sigma)$
(WHILE)	$(\textbf{while}\ e_0\ \textbf{do}\ e_1, \sigma) \to (\textbf{if}\ e_0\ \textbf{then}\ e_1\,;\, \textbf{while}\ e_0\ \textbf{do}\ e_1, \sigma)$

sowie den entsprechenden exception-Regeln und den durch Speicherzustände erweiterten small step Regeln von $\mathcal{L}_3$ herleiten lässt.

3.5.3 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.

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

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

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

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.

3.5.4 Syntaktischer Zucker

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

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

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

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.

Kapitel 4

Abschließende Bemerkungen

Nach der Beschreibung des Programms und des Regelwerks sollen hier noch ein paar abschließende Bemerkungen folgen.

4.1 Bugs

Wie es in der Natur von Software mit einer gewissen Komplexität liegt, ist auch TPML nicht perfekt und es können immer wieder bisher nicht entdeckte Fehler zu Tage treten. Diese sollten dann in der nächsten Übung angesprochen werden, so dass sie möglichst schnell beseitig werden können und eine neue Version verfügbar gemacht werden kann, ohne diese Fehler.

Aus Sicht des Studierenden sollte man dies vielleicht sogar als Chance verstehen, denn wenn man einen Fehler in einer Regel findet, heisst das in der Regel, dass man die Regel verstanden hat.

Anhang A

Lizenz

Das vollständige Programm steht im Quelltext zur Verfügung, unter der MIT Lizenz.

```
Copyright (c) 2005-2006 University of Siegen

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the ‘‘Software’’), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ‘‘AS IS’’, WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
```

Die Vervielfältung und Anpassung des Programms ist somit ohne Einschränkungen möglich, und es somit insbesondere auch anderen Universitäten und Fach-

hochschulen erlaubt TPML in einer ggfs. angepassten Version als interaktives Lernwerkzeug für die Lehre im Bereich ,,Theorie der Programmierung" einzusetzen.

Bei Fragen wenden Sie sich bitte an die Fachgruppe ,,Theoretische Informatik" der Universität Siegen.

Stichwortverzeichnis