TPML 2.0
Benutzerhandbuch

Christian Fehler, Christoph Fehling, Marcell Fischbach,
Benedikt Meurer, Benjamin Mies, Michael Oeste
Fachbereich 12 - Informatik und Elektrotechnik
Universität Siegen

19. Oktober 2007

Zusammenfassung

Die Vorlesung „*TheoriederProgrammierung*" 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

1 Einleitung **6**

1.1 Systemvoraussetzungen 6

1.2 Installation 7

1.3 Erste Schritte 7

2 Benutzerinterface **9**

2.1 Überblick 9

2.2 Quelltexteditor 9

2.2.1 Auto-Vervollständigung 10

2.3 Outline 11

2.3.1 Einstellungen 12

2.4 PDF-Export 13

2.5 LaTeX-Export 14

2.5.1 Der LaTeX-Export Dialog 14

2.6 Beweiswerkzeuge 15

2.6.1 Mouse-Over-Effekt 15

2.7 Tutorials 16

2.7.1 Small Step Interpreter 17

2.7.2 Big Step Interpreter 18

2.7.3 Type Checker 19

2.7.4 Type Inference 19

2.7.5 Minimal Typing 22

2.7.6 Sub Typing 22

3 Die Sprachen im Detail 23

3.1 Die Sprache $\mathcal{L}_0$ 23

3.1.1 Big step Semantik von $\mathcal{L}_0$ 24

3.1.2 Small step Semantik von $\mathcal{L}_0$ 24

3.2 Die Sprache $\mathcal{L}_0^{CBN}$ 25

3.2.1 Big step Semantik von $\mathcal{L}_0^{CBN}$ 25

3.2.2 Small step Semantik von $\mathcal{L}_0^{CBN}$ 25

3.3 Die Sprache $\mathcal{L}_1$ 26

3.3.1 Big step Semantik von $\mathcal{L}_1$ 27

3.3.2 Small step Semantik von $\mathcal{L}_1$ 29

3.3.3 Typechecker Semantik von $\mathcal{L}_1$ 30

3.3.4 Minimal Typing Semantik von $\mathcal{L}_1$ 31

3.3.5 Syntaktischer Zucker 32

3.4 Die Sprache $\mathcal{L}_1^{CBN}$ 33

3.4.1 Big step Semantik von $\mathcal{L}_1^{CBN}$ 33

3.4.2 Small step Semantik von $\mathcal{L}_1^{CBN}$ 33

3.5 Die Sprache $\mathcal{L}_1^{SUB}$ 34

3.5.1 Sub Typing Semantik von $\mathcal{L}_1^{SUB}$ 34

3.5.2 Rec Sub Typing Semantik von $\mathcal{L}_1^{SUB}$ 34

3.6 Die Sprache $\mathcal{L}_2$ 35

3.6.1 Big step Semantik von $\mathcal{L}_2$ 35

3.6.2 Small step Semantik von $\mathcal{L}_2$ 36

3.6.3 Typechecker Semantik von $\mathcal{L}_2$ 36

3.6.4 Minimal Typing Semantik von $\mathcal{L}_2$ 36

3.6.5 Syntaktischer Zucker 36

3.7 Die Sprache $\mathcal{L}_2^{CBN}$ 37

3.7.1 Big step Semantik von $\mathcal{L}_2^{CBN}$ 37

3.7.2 Small step Semantik von $\mathcal{L}_2^{CBN}$ 38

3.8 Die Sprache $\mathcal{L}_2^{SUB}$ 38

3.9 Die Sprache $\mathcal{L}_2^O$ . . . 38
3.9.1 Big step Semantik von $\mathcal{L}_2^O$ . . . 39
3.9.2 Small step Semantik von $\mathcal{L}_2^O$ . . . 40
3.9.3 Typechecker Semantik von $\mathcal{L}_2^O$ . . . 41
3.9.4 Minimal Typing Semantik von $\mathcal{L}_2^O$ . . . 42
3.9.5 Syntaktischer Zucker . . . 42
3.10 Die Sprache $\mathcal{L}_2^{OSUB}$ . . . 43
3.10.1 Sub Typing Semantik von $\mathcal{L}_2^{OSUB}$ . . . 43
3.10.2 Rec Sub Typing Semantik von $\mathcal{L}_2^{OSUB}$ . . . 43
3.11 Die Sprache $\mathcal{L}_2^C$ . . . 44
3.11.1 Big step Semantik von $\mathcal{L}_2^C$ . . . 44
3.11.2 Small step Semantik von $\mathcal{L}_2^C$ . . . 45
3.12 Die Sprache $\mathcal{L}_3$ . . . 46
3.12.1 Big step Semantik von $\mathcal{L}_3$ . . . 47
3.12.2 Small step Semantik von $\mathcal{L}_3$ . . . 47
3.12.3 Typechecker Semantik von $\mathcal{L}_3$ . . . 48
3.12.4 Minimal Typing Semantik von $\mathcal{L}_3$ . . . 50
3.12.5 Syntaktischer Zucker . . . 50
3.13 Die Sprache $\mathcal{L}_3^{SUB}$ . . . 51
3.13.1 Sub Typing Semantik von $\mathcal{L}_3^{SUB}$ . . . 51
3.13.2 Rec Sub Typing Semantik von $\mathcal{L}_3^{SUB}$ . . . 51
3.14 Die Sprache $\mathcal{L}_4$ . . . 52
3.14.1 Big step Semantik von $\mathcal{L}_4$ . . . 52
3.14.2 Small step Semantik von $\mathcal{L}_4$ . . . 53
3.14.3 Typechecker Semantik von $\mathcal{L}_4$ . . . 54
3.14.4 Syntaktischer Zucker . . . 54
3.15 Die Sprache $\mathcal{L}_4^{SUB}$ . . . 55
3.15.1 Sub Typing Semantik von $\mathcal{L}_4^{SUB}$ . . . 55
3.15.2 Rec Sub Typing Semantik von $\mathcal{L}_4^{SUB}$ . . . 55
3.16 Konkrete Syntax . . . 55

4 Abschließende Bemerkungen 57

4.1 Bugs 57

A Lizenz 58

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 Systeme, wie zum Beispiel Mac OS X Tiger, diverse Linux Distributionen und angepasste Windows OEM Installationen, 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 Menü 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.

Sollte die Versionskontrolle einmal versagen, kann sie auch übergangen werden. Dazu muss die JAR-Datei des UI-Paketes manuell gestartet werden.

```
java -jar de.unisiegen.tpml.ui-X.Y.Z.jar -f
```

1.3 Erste Schritte

Nach dem Start des Programms können neue Dateien erstellt werden. Dateien sind immer mit einer Programmiersprache verknüpft, die durch die

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

Dateiendung identifiziert wird[2]. In TPML wurden die Sprachen $\mathcal{L}_0$ bis $\mathcal{L}_4$ realisiert, die den wesentlichen in der Vorlesung behandelten Sprachen entsprechen, wobei die Namen und der Umfang einzelner Sprachen von denen in der Vorlesung abweichen kann. Die Sprachen werden im Detail in Kapitel 3 behandelt, welches auch als Nachschlagewerk dienen soll.

Um eine neue Datei zu erstellen, wählt man aus dem Menü **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 $\mathcal{L}_0$. Die Sprachen sind in Kategorien unterteilt, $\mathcal{L}_0$ ist natürlich in der ersten ganz oben zu finden. Nach der Auswahl der Programmiersprache öffnet sich ein Quelltexteditor, 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 bemerken wird, unterstützt der Editor Syntaxhighlighting und markiert automatische Fehler im Programmtext, wie man es in der Regel von integrierten Entwicklungsumgebungen gewohnt ist[3].

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.

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

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

Kapitel 2

Benutzerinterface

2.1 Überblick

Das Menü **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 2.6 genauer betrachtet. Mit **PDF Export** lässt sich der derzeitig geführte Beweis bzw. der editierte Quellcode einfach in eine PDF Datei speichern. Siehe dazu auch Kapitel 2.4. Über **Latex Export** kann man dies in eine Tex-Datei exportieren. Mehr dazu im Kapitel 2.5. Mit **Einstellungen...** können die Einfärbung von Schlüsselworten im Quelltexteditor 2.2 und andere Farben eingestellt werden. Das Menü **Beweis** enthält die einzelnen Beweiswerkzeuge und dient dazu, zwischen dem **Beginner** und **Fortgeschrittener** Modus zu wechseln. Diese Modi beeinflussen die bei den Beweisschritten zur Verfügung stehenden Regeln und die Ansicht der verschiedenen Modi.

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 den Ausdruck kopieren. Alternativ kann auch die Dateiendung einer gespeicherten Datei geändert werden.

Ist ein eingegebener Ausdruck fehlerhaft, 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 Tooltip, der den Fehler genauer beschreibt. Ein Klick auf das Error-Icon bewirkt, dass der fehlerhafte Text markiert wird. Wird nun noch einmal auf das Error-Icon geklickt, wird der markierte Text entfernt.

Der Quelltexteditor hebt freie Identifier und Type-Namen in der eingestellten Farbe hervor, da freie Identifier oder Type-Namen meist auf eine fehlerhafte Eingabe zurückzuführen sind. Gibt der Anwender zum Beispiel „**let** $x = a$ **in** x“ ein, ist dies natürlich nicht falsch, aber der Type Checker würde den Ausdruck nicht akzeptieren, da das „a“ in dem Ausdruck frei vorkommend ist.

2.2.1 Auto-Vervollständigung

Der Quelltexteditor besitzt eine Auto-Vervollständigung. Gibt der Benutzer den ersten Teil eines Ausdrucks oder Typs ein, erscheint in der betroffenen Zeile ein gelbes Dreieck mit einem Ausrufezeichen. Bewegt man den Mauszeiger über das Icon, so erscheint ein Tooltip, der einen Vorschlag angibt, was der Benutzer noch eingeben muss, um die Eingabe zu vervollständigen. Der Tooltip gibt ebenfalls an, um welchen Ausdruck oder Typ es sich handelt, was besonders für Ungeübte eine wertvolle Information ist.

Der Ausdruck, der noch vervollständigt werden muss, wird im Hintergrund hervorgehoben, so dass der Anwender genau erkennen kann, welchen Teil er noch vervollständigen muss. Gibt der Anwender „$1 +$ **let** $x =$“ ein, wird der Teil „**let** $x =$“ hervorgehoben und der Tooltip verrät dem Anwender, dass er noch „e_1 **in** e_2“ eingeben muss um „**Let**“ zu vervollständigen.

Der Anwender hat die Möglichkeit auf das Icon zu klicken, dadurch wird der Vorschlag automatisch an der richtigen Stelle eingefügt. Im gegebenen Beispiel wird der Ausdruck „$1+$**let** $x =$“ automatisch zu „$1+$**let** $x = e1$ **in** $e2$“ erweitert, was für den Parser einen gültigen Ausdruck darstellt. Die Auto-Vervollständigung ist ebenfalls in der Lage, Fehler innerhalb eines Ausdrucks zu erkennen und durch einen Klick auf das Icon zu beheben.

Des weiteren bietet der Parser die Möglichkeit, Identifier automatisch umzubenennen, wenn durch die Namen der Identifier ein Konflikt entstanden ist. Dies kann bei Klassen der Fall sein, ein Beispiel hierfür ist:

„**class** $(self)$ **inherit** a **from** (**class** $(self)$ **val** $a = 0$; **end**) ; **val** $a = 1$; **method** $m = a$; **end**“

In diesem Beispiel besteht ein Konflikt zwischen dem Identifier „a“ im ersten Teil des „**inherit**“ und dem Attribut in der zweiten Reihe. Dieser Konflikt würde im Small Step Interpreter zu dem Ergebniss führen, dass eine Reihe entstehen würde, die zweimal das Attribut „a“ enthält, deshalb lehnt der Parser diesen Ausdruck ab. Der Konflikt kann aber dadurch gelöst werden, dass das Attribut umbenannt wird. Dabei muss darauf geachtet werden, dass auch alle an den Identifier des Attributes gebundenen Identifier umbenannt werden. Dies betrifft in diesem Fall das „a“ im Rumpf der Methode. Der Parser blendet in diesem Fall ein blaues Fehlersymbol ein, das dem Anwender signalisieren soll, dass eine Umbenennung möglich ist. Klickt der Anwender auf das Symbol, werden die Identifier umbenannt und es entsteht folgender Ausdruck:

„**class** $(self)$ **inherit** a **from** (**class** $(self)$ **val** $a = 0;$ **end**) ; **val** $a' = 1;$ **method** $m = a'$; **end**“

2.3 Outline

Die Outline erlaubt es dem Anwender, eingegebene Ausdrücke und Typen in ihrer Baumansicht zu betrachten. Der Anwender ist somit in der Lage zu erkennen, aus welchen Teilen ein Ausdruck oder Typ besteht. So ist zum Beispiel bei dem Ausdruck „1+1+1“ nicht auf den ersten Blick ersichtlich, wie sich die Applikationen zusammensetzen. In der Outline wird sichtbar, dass der Ausdruck als wie folgt geklammert gesehen werden kann: „$(1 + 1) + 1$“. Dieses Resultat ist nicht besonders überraschend, betrachtet man aber den Funktions-Typen „**int** → **int** → **int**“, so ergibt sich folgende Klammerung „**int** → (**int** → **int**)“. Der Anwender kann darüber hinaus erkennen, um welche Ausdrücke es sich handelt, da der Name in der Outline dargestellt wird. Vor dem Namen des Ausdrucks steht immer die Menge zu der er gehört. Diese Menge ist zumeist „**e**“ oder „**v**“, falls es sich um einen Wert handelt. Es sind aber für Identifier, Reihen, Typen etc. auch viele andere Mengen vorhanden. Neben der Menge, zu der ein Ausdruck oder Typ gehört, wird auch sein Index dargestellt. Dadurch ist erkennbar, aus welchen Teilen sich ein Ausdruck oder Typ zusammensetzt. Zum Beispiel setzt sich „**let** $id = 1 + 1$ **in** $id + 2$“ aus dem Identifier „id“, aus einem „e_1“ und einem „e_2“ zusammen.

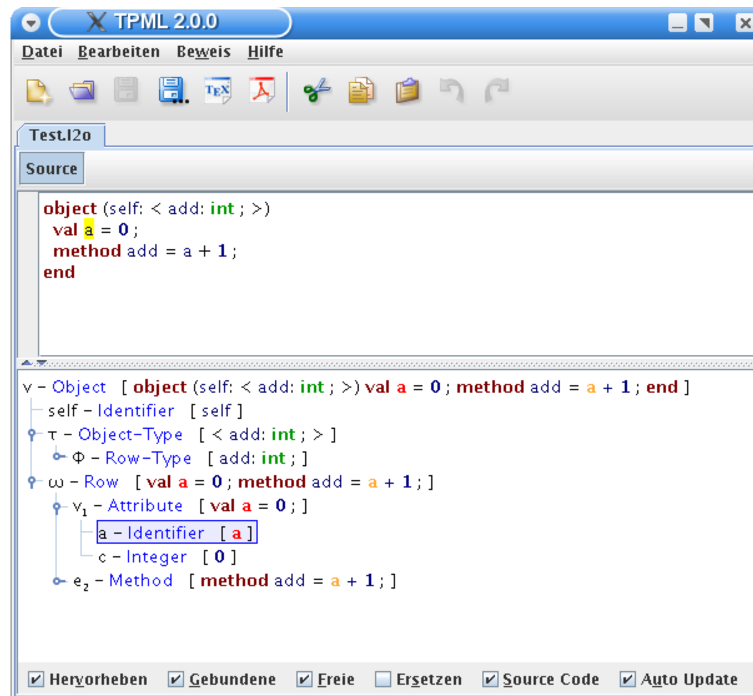

Abbildung 2.1: Outline

2.3.1 Einstellungen

Die Outline bietet verschiedene Einstellungsmöglichenkeiten, die nun kurz angesprochen werden sollen.

Hervorheben bietet die Möglichkeit, den aktuell in der Outline selektierten Knoten in höheren Knoten zu markieren. Ist zum Beispiel der Ausdruck „**let** $x = 1 + 1$ **in** x“ in der Outline geladen und der Knoten „$1 + 1$“ selektiert, wird der zweite Ausdruck in dem ersten hervorgehoben. Die Farbe der Hervorhebung kann, wie alle anderen Farben auch, in den Einstellungen von TPML verändert werden.

Gebundene steht dafür, dass gebundene Identifier und Typ-Namen, wenn sie in der Outline selektiert werden, in höheren Knoten hervorgehoben werden. Der Ausdruck „$\lambda x.x + 1$“ besteht aus einen Identifier „x“ und aus einem Kind-Ausdruck „$x + 1$“. Selektiert man nun den Identifier, werden bei aktiver Option alle Identifier hervorgehoben, die an diesen Identifier gebunden sind. Dies funktioniert auch umgekehrt, wird also das „x“ im Ausdruck „$x + 1$“ selektiert, wird der Identifier hervorgehoben, an den dieser Identifier gebunden ist. Vor allem bei komplizierteren Ausdrücken mit gleichen Identi-

fiern wie „**let** $x = 1$ **in** $x +$ **let** $x = 2$ **in** x“ können so die Bindungen durch die Outline überprüft werden.

Wenn **Freie** aktiv ist, werden freie Identifier und Typ-Namen in der Outline hervorgehoben. Dies ist besonders nützlich, da freie Identifier oder Typ-Namen meist auf eine fehlerhafte Eingabe zurückzuführen sind. Gibt der Anwender zum Beispiel „**let** $x = a$ **in** x“ ein, ist dies natürlich nicht falsch, aber der Type Checker würde den Ausdruck nicht akzeptieren, da das „a“ in dem Ausdruck frei vorkommend ist. Es werden allerdings nur Identifier und Typ-Namen hervorgehoben, die in dem Wurzelknoten frei vorkommend sind, und nicht alle in den jeweiligen Knoten frei vorkommenden Identifier oder Typ-Namen.

Die Option **Kompakt** steht dafür, dass der selektierte Ausdruck in höheren Knoten durch „...“ ersetzt wird. Dadurch werden höhere Knoten kompakter dargestellt, was bei größeren Ausdrücken von Vorteil ist.

Ist die Option **Source Code** aktiv, wird der Source Code im jeweiligen Quelltexteditor hervorgehoben, der dem selektierten Knoten in der Outline entspricht. Ist diese Option nicht aktiv, kann der Source Code durch einen Doppelklick auf den Outline Knoten ebenfalls hervorgehoben werden. Diese Option ist nur bei den Quelltexteditoren verfügbar und nicht bei den verschiedenen Beweiswerkzeugen.

Auto Update steht dafür, dass die Outline automatisch geladen wird, wenn sich etwas ändert. Dies geschieht nach einer gewissen Verzögerung, die nötig ist, damit sich nicht zu viele Updates bei einer Benutzereingabe berechnet werden müssen. Diese Option ist nur in Ansichten aktiv, in denen Änderungen eindeutig sind, also in den Quelltexteditoren und im Small Step Interpreter. In den anderen Beweiswerkzeugen können bei einem Schritt mehrere Knoten erzeugt werden, und ein Laden der Outline wäre nicht eindeutig möglich. Ist die Option nicht aktiv oder gar nicht verfügbar, kann der Anwender auf die entsprechenden Ausdrücke oder Typen klicken, und diese werden dann geladen. Gleiches gilt für die Quelltexteditoren: Klickt man mit der Maus auf den Editor, wird der aktuelle Ausdruck oder Typ in die Outline geladen. Ist im Quelltexteditor im Moment kein gültiger Ausdruck geladen, wird die Outline rot umrandet.

2.4 PDF-Export

Über den PDF-Export lassen sich Quelltext und Beweisschritte in eine PDF-Datei exportieren, die optisch nahezu der Progammausgabe entsprechen. Um

einen Quelltext oder eine Beweisführung in eine PDF-Datei zu exportieren wählt man einfach den entsprechenden Menüeintrag unter **Datei** aus oder klickt auf den entsprechenden Button in der Symbolleiste. Der Benutzer hat nun die Wahl zwischen dem Hoch- und dem Querformat. Letzerer kann bei längeren Ausdrücken sinnvoll sein.

2.5 LaTeX-Export

Über den LaTeX-Export lassen sich Quelltext und Beweisschritte in eine TeX-Datei exportieren. Welche dann in eine PDF Datei mit Syntaxhighlighting überführt werden kann. Bei dem Quelltext gibt es die Einschränkung, dass nur gültige Ausdrücke exportiert werden können. Erstellt man ein PDF aus exportierten Beweisschritten, so erhält man eine Optik, welche stark an die TPML Ausgabe angelehnt ist. Weiterhin besteht die Möglichkeit sich über den Menüeintrag **TPML.TEX erstellen...** eine Datei generieren zu lassen, welche alle Commands, die in TPML verwendet werden, enthält.

Um ein pdf zu erstellen können die folgenden drei Befehle verwendet werden:

latex *filename.tex* (2 mal ausführen)

dvips *filename.dvi*

ps2pdf *filename.ps*

Damit das Konvertieren fehlerfrei funktioniert, werden die Pakete **amsmath**, **amssymb**, **amstext**, **color**, **ifthen**, **longtable**, **pst-node** und **pstricks** benötigt.

Unter Windows kann man zum Beispiel das Tool **MIKTEX** verwenden, welches auf der Seite http://miktex.org/ zum Download bereit steht.

2.5.1 Der LaTeX-Export Dialog

Es gibt noch verschiedene Einstellmöglichkeiten, welche man vornehmen kann. Zunächst einmal ist der Name der Zieldatei frei wählbar. Desweiteren kann man sich entscheiden, ob man eine einzige Zieldatei mit allen benötigten Commands erstellen möchte, oder die Commands aus einer separaten Datei verwendet werden sollen (tpml.tex), welche sich beim compilieren im selben Ordner befinden muss.

Bei den Beweismethoden, welche in einer Baumstruktur dargestellt werden, hat man die Möglichkeit eine Überlappung der Seiten in *mm* anzugeben (Standard ist keine Überlappung) und die Anzahl der Seiten anzugeben, da 5 Seiten voreingestellt sind. Werden weniger Seiten benötigt, entstehen am Ende eine entsprechende Anzahl an weißen Seiten, welche durch die eigene Angabe einer Seitenanzahl entfernt werden können.
Leider ist es nicht möglich ein Dokument in Baumstruktur über Latex in ein pdf zu konvertieren, welches die Grenze von 13 Seiten überschreitet. Dies kann allerdings über den Standard PDF-Export erzeugt werden.

2.6 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, ein Type Checker usw. aktiv sein. Diese können über die entsprechende Schaltflächen angewählt werden. Natürlich ergibt es keinen Sinn, Beweiswerkzeuge, die mit dem Typsystem zu tun haben, bei Dateien zu starten, die kein Typsystem haben, wie beispielsweise $\mathcal{L}_0$. 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 Menü **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, die zum Beweis des Ausdrucks zur Verfügung stehen, und teilweise in der Ansicht des Beweiswerkzeuges. So werden beispielsweise beim Typ Checker im **Beginner** Typen erst dann angezeigt, wenn der Typ bewiesen ist.

2.6.1 Mouse-Over-Effekt

Jede der Beweismethoden hat für den Benutzer eine weitere Komfort-Funktion eingebaut. Bei größeren Programmen, die bewiesen werden sollen, und insbesondere, wenn der gleiche Identifier doppelt benutzt wird, verliert man schnell die Übersicht. Folgendes Beispiel soll dies verdeutlichen:

$\textbf{let}\ x = \lambda x.x\,x\ \textbf{in}\ x\,x$

Wie man leicht sieht, ist der Identifier „x“ doppelt verwendet. In noch größeren Beispielen wäre es nicht mehr einfach zu überschauen, an welches „x“ welches gebunden ist. Um dies zu vereinfachen kann man einfach mit dem Mauszeiger über eines der „x“ fahren, und die dazugehörigen „x“ werden farbig hervorgehoben. Es werden zwei verschiedene Farben [1] benutzt, je nachdem, ob es sich bei dem Identifier um den bindenden Identifier handelt, oder um einen, der an einen anderen gebunden ist. Im gegebenen Beispiel würde das „x“ nach dem „let“ zusammen mit den letzten beiden „x“ hervorgehoben.

Identifier, die keine Bindungen haben, aber Bindungen haben könnten, werden ebenfalls hervorgehoben. Anders Identifier, die gar keine Bindungen haben können, wie es bei Namen für Methoden der Fall ist. Dazu sei ein letztes Beispiel zu diesem Thema gegeben:

object $(self)$ **method** $inc\ x\ y = x + 1$; **end**

Dieses Beispiel hat drei Identifier: inc, x und y. x verhält sich wie im oberen Beispiel. Es wird zusammen mit dem gebundenen x hervorgehoben. y dagegen hat zwar keinen weiteren gebundenen Identifier, könnte aber einen solchen haben. Er wird allein hervorgehoben. inc letzlich wird nicht hervorgehoben. Es ist ein Methodenname, und an den kann prinzipiell nichts gebunden sein.

2.7 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 vorraussichtlich das Produkt aus 3 und 4 berechnet.

---

[1] Die Farben, sowohl für gebundenen als auch bindende Identifier können über Einstellungen... im Menü Bearbeiten eingestellt werden.

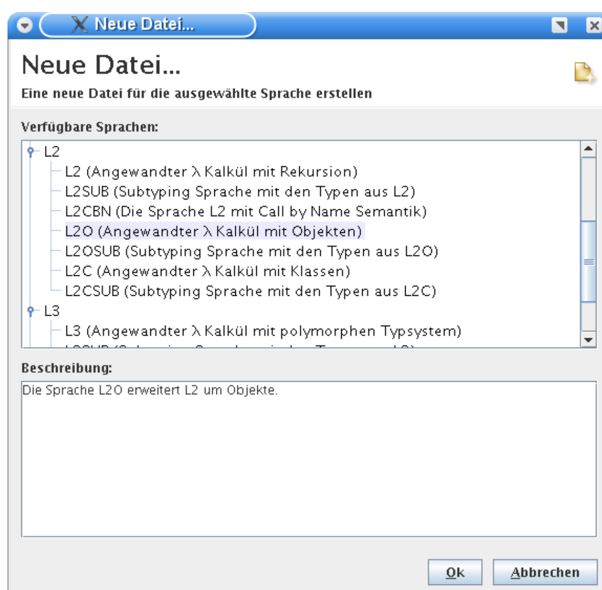

Abbildung 2.2: Eine neue Datei erstellen

2.7.1 Small Step Interpreter

Unsere mutige Behauptung wollen wir nun natürlich durch einen stichhaltigen Beweis untermauern. Wir starten also mittels **Beweis → Small Step** oder mit der Taste F9 einen Small Step Interpreter.

Bewegt man nun den Mauscursor über den Buttons des Regelmenüs oberhalb des Pfeils für den ersten Ableitungsschritt, wird der Teil des Ausdrucks rot unterstrichen, der als nächstes abgeleitet werden soll, wie in Abbildung 2.3 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 Regelmenü 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 Regelmenü. Eine komplette Beweisführung wird mittels **Vervollständigen** im selben Menü vorgenommen.

Ist man mit den Regeln vertraut, so kann man über **Beweis → Fortgeschrittener** einige Regeln ausblenden. Das Regelmenü 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 wählt lediglich **COND-TRUE** bzw. **COND-FALSE**

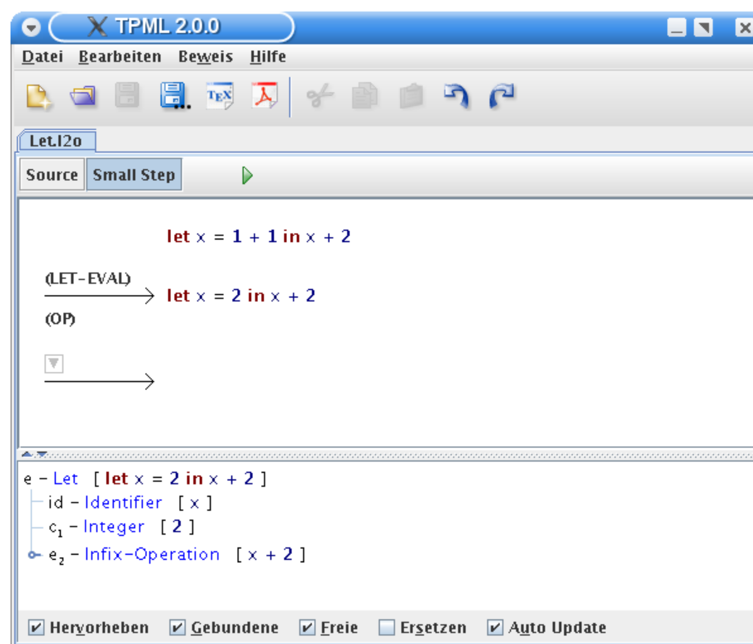

Abbildung 2.3: Der Small Step Interpreter

aus.

2.7.2 Big Step Interpreter

Der Big Step Interpreter kann auch über das **Beweis** Menü oder mit der Taste F11 gestartet werden. Man muss sich hierfür nicht zwingend im Quelltexteditor befinden. Auch beim Big Step Interpreter können Regeln über das Regelmenü 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.

2.7.3 Type Checker

Diese Beweisart ist genau wie der Big Step Interpreter in einer Baumstruktur aufgebaut. Der Type Checker verhält sich daher bezüglich des Anwendens 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 Regelmenü selbst einen Typ für den entsprechenden Teilbaum eingeben, wie in Abbildung 2.4 gezeigt. Der Typinferenzalgorithmus versucht dann den Unterbaum zu diesem Typ auszuwerten.

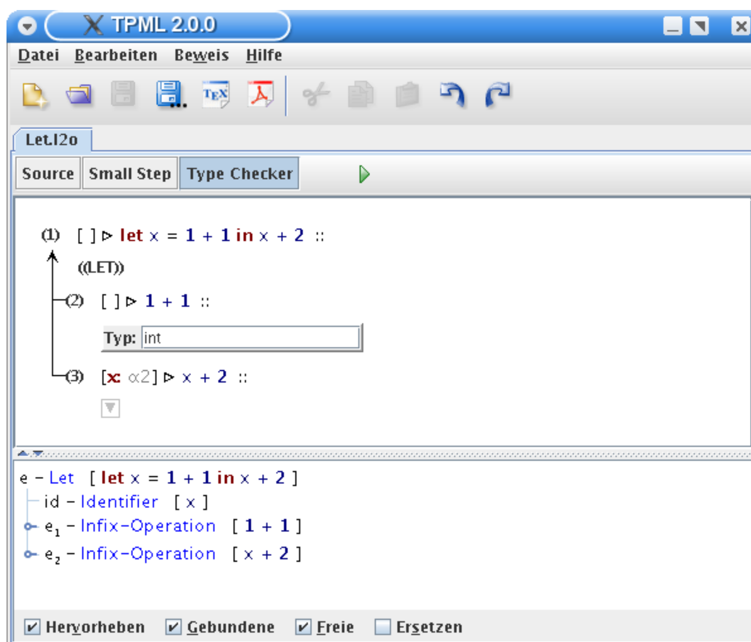

Abbildung 2.4: Der Type Checker

2.7.4 Type Inference

Der Type Inference Algorithmus ist, wie der Small Step Interpreter auch, in einer Listenstruktur realisiert und kann über **Beweis** → **Type Inference** gestartet werden. Die Bedienung ist auch diesmal wieder an die bereits genannten Beweisarten angelehnt. Es besteht auch hier die Möglichkeit, selbst

Regeln auf den aktuellen Ausdruck anzuwenden, den nächsten Schritt erraten zu lassen, oder den kompletten Ausdruck zu vervollständigen.

Wendet man die Regel Unify an, werden die evtl. daraus hervorgehenden Type Substitutions in einer Liste über den Type Formulas gesammelt. Dabei ist die zuletzt gesammelte Type Substitution sichtbar. Die bereits vorher gesammelten Type Substitutions werden durch drei Punkte angedeutet, und können als Tooltip über den Punkten eingesehen werden.

Wenn man sich etwas mit der Funktionsweise des Algorithmus vertraut gemacht hat, kann man über **Beweis → Fortgeschrittener** in den Advanced Modus wechseln. In diesem Modus werden Type Equations, bei denen beide Typvariablen aus Arrow Types bestehen, jetzt nicht mehr neue Type Equations aufgenommen, sondern diese werden direkt zu Type Substitutions aufgelöst. Außerdem wird beim Anwenden einer Regel auf den Ausdruck nicht mehr vom System versucht, die Regel einer Type Formula zuzuordnen, sondern es wird versucht, die Regel auf die erste Type Formula anzuwenden. Diese können allerdings durch Drag and Drop umsortiert werden.

Das Regelwerk ist, mit einer Ausnahme, äquivalent zum Type Checker. Es wurde noch die Regel **UNIFY** hinzugenommen. Die sich aus folgenden Ableitungsschritten zusammensetzt:

(EMPTY) $\mathit{unify}(A, \emptyset) = [\,]$

(ASSUME) $\mathit{unify}(A, \{\tau = \tau'\} \cup E) = \mathit{unify}(A, E)$ falls $(\tau = \tau') \in A$

(TRIV) $\mathit{unify}(A, \{\tau = \tau\} \cup E) = \mathit{unify}(A, E)$

(MU-LEFT) $\mathit{unify}(A, \{\mu t\tau = \tau'\} \cup E) = \mathit{unify}(A, \{\tau[\mu t\tau/t] = \tau'\} \cup E)$

(MU-RIGHT) $\mathit{unify}(A, \{\tau = \mu t\tau'\} \cup E) = \mathit{unify}(A, \{\tau = \tau'[\mu t\tau'/t]\} \cup E)$

(VAR) $\mathit{unify}(A, \{\alpha = \tau\} \cup E) = \mathit{unify}(A, \{\tau = \alpha\} \cup E)$

$$= \begin{cases} [\tau/\alpha] \circ s & \text{falls } \alpha \notin \mathit{var}(\tau) \text{ und } \mathit{unify}(E[\tau/\alpha]) = s \\ \text{“nicht lösbar”} & \text{falls } \alpha \notin \mathit{var}(\tau) \text{ und} \\ & \mathit{unify}(E[\tau/\alpha]) = \text{“nicht lösbar”} \\ & \text{oder } \alpha \in \mathit{var}(\tau) \text{ und } \alpha \neq \tau \end{cases}$$

(ARROW) $\mathit{unify}(A, \{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'\} \cup E)$
$= \mathit{unify}(A \cup \{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'\}, \{\tau_1 = \tau_1',\ \tau_2 = \tau_2'\} \cup E)$

(TUPLE) $\mathit{unify}(A, \{\tau_1 * \ \ldots\ * \tau_n = \tau_1' * \ \ldots\ * \tau_n'\} \cup E)$
$= \mathit{unify}(A \cup \{\tau_1 * \ \ldots\ * \tau_n = \tau_1' * \ \ldots\ * \tau_n'\},$
$\{\tau_1 = \tau_1',\ \ldots\ , \tau_n = \tau_n'\} \cup E)$

(LIST) $\mathit{unify}(A, \{\tau\,\mathbf{list} = \tau'\,\mathbf{list}\} \cup E)$
$= \mathit{unify}(A \cup \{\tau\,\mathbf{list} = \tau'\,\mathbf{list}\}, \{\tau = \tau'\} \cup E)$

(REF) $\mathit{unify}(A, \{\tau\,\mathbf{ref} = \tau'\,\mathbf{ref}\} \cup E)$
$= \mathit{unify}(A \cup \{\tau\,\mathbf{ref} = \tau'\,\mathbf{ref}\}, \{\tau = \tau'\} \cup E)$

(OBJECT) $\mathit{unify}(A, \{< \phi > = < \phi' >\} \cup E)$
$= \mathit{unify}(A \cup \{< \phi > = < \phi' >\}, \{\phi = \phi'\} \cup E)$

(ROW) $\mathit{unify}(A, \{(m_1 = \tau_1\ ;\ \ldots\ ;\ m_n = \tau_n;\ \phi)$
$= (m_1 = \tau_1'\ ;\ \ldots\ ;\ m_n = \tau_n';\ \phi')\} \cup E)$
$= \mathit{unify}(A \cup \{(m_1 = \tau_1\ ;\ \ldots\ ;\ m_n = \tau_n;\ \phi)$
$= (m_1 = \tau_1'\ ;\ \ldots\ ;\ m_n = \tau_n';\ \phi')\},$
$\{\tau_1 = \tau_1',\ \ldots\ , \tau_n = \tau_n', \phi = \phi'\} \cup E)$

(STRUCT) $\mathit{unify}(A, \{\tau_1 = \tau_2\} \cup E) =$ “nicht lösbar”
in allen anderen Fällen

Bei der Regel **ROW** müssen die Methodennamen nicht in der gleichen Reihenfolge vorkommen. Zudem müssen die Methodennamen nicht in beiden

Reihen vorkommen, da für den Fall, dass ϕ nicht gleich ϵ ist, *unify* auch mit ϕ und den restlichen Methoden von τ' aufgerufen werden kann. Zum Beispiel führt $unify(A, \{(b : \alpha_1; \alpha_2) = (a : int; b : int;)\} \cup E)$ zu dem Aufruf von $unify(A \cup \{(b : \alpha_1; \alpha_2) = (a : int; b : int;)\}, \{\alpha_1 = int, \alpha_2 = a : int\} \cup E)$.

2.7.5 Minimal Typing

Der Minimal Typing Algorithmus ist, wie der Big Stepper und der Type Checker, in einer Baumstruktur realisiert. Auch die Bedienung dieser Beweismethode ist äquivalent zu den beiden anderen. Es besteht auch hier die Möglichkeit, die nächste Regel über das Pull Down Menü auszuwählen, einen Schritt vom Algorithmus selbst erraten zu lassen, oder den kompletten Beweis vervollständigen zu lassen. Für den Fall, dass der Ausdruck synktaktischen Zucker enthält, kann auch hier in Kernsyntax übersetzt werden.

2.7.6 Sub Typing

Zur Überprüfung von Subtyprelationen wurde ein eigener Source Editor angelegt. Dieser beinhaltet zwei Eingabefelder, welche von Bedienung und Aussehen dem normalen Source Editor gleichen, allerdings nur für die Eingabe von Typen gedacht sind und nicht für ganze Expressions. Das erste Eingabefeld soll den Subtyp entgegennehmen und das zweite Eingabefeld den Supertyp, für welche man die Subtyp Realation überprüfen will. Ein weiterer Unterschied zum allgemeinen Source Editor ist, dass man eine Outline pro Eingabefeld hat, statt wie vorher insgesamt eine Outline.

Für den Beweis hat man die Wahl zwischen einfachem Sub Typing, und Sub Typing mit rekursiven Typen. Beide sind von Bedienung und Aussehen her exakt gleich. Auch hier wurde bei der Visualisierung wieder eine Baumstruktur verwendet, und von der Bedienung gleicht Subtyping den schon bekannten Beweismethoden.

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 $\mathcal{L}_n$. Zusätzlich dazu gibt es die Sprache $\mathcal{L}_2^O$, welche die Sprache $\mathcal{L}_2$ um objektorientierte Konzepte erweitert und die Sprache $\mathcal{L}_2^C$, welche die Sprache $\mathcal{L}_2^O$ um Klassen mit Vererbung erweitert. Zu den Sprachen $\mathcal{L}_0$ bis $\mathcal{L}_2$ gibt es noch die Varianten mit Call by Name Semantik $\mathcal{L}_0^{CBN}$ bis $\mathcal{L}_2^{CBN}$, die ausgewählt werden können. Es existieren zu allen Sprachen, die ein Typsystem enthalten die entsprechenden Subtyping Sprachen $\mathcal{L}_n^{SUB}$. 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*).

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

Die abstrakte Syntax enthält lediglich drei Produktionen.

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

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

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

definiert.

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

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

(VAL) $\quad v \Downarrow v$

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

(APP) $\quad \dfrac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1\, v_2 \Downarrow v}{e_1\, e_2 \Downarrow v}$

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

(BETA-V) $\quad (\lambda id.e)\, v \to e[v/id]$

(APP-LEFT) $\quad \dfrac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2}$

(APP-RIGHT) $\quad \dfrac{e \to e'}{v\, e \to v\, e'}$

herleiten lässt.

3.2 Die Sprache $\mathcal{L}_0^{CBN}$

Die Sprache $\mathcal{L}_0^{CBN}$ bietet den gleichen Funktionsumfang wie die Sprache $\mathcal{L}_0$, benutzt aber statt der Call by Value eine Call by Name Semantik. Um dies zu realisieren werden verschiedene Regeln in der Big step und der Small step Semantik gelöscht oder geändert. Nur diese geänderten bzw. gelöschten Regeln werden hier aufgeführt.

3.2.1 Big step Semantik von $\mathcal{L}_0^{CBN}$

Ein big step heißt gültig für $\mathcal{L}_0^{CBN}$, wenn er sich mit den big step Regeln von $\mathcal{L}_0$ und den geänderten bzw. gelöschten Regeln

(BETA-V) nicht vorhanden

(BETA) $\dfrac{e_1[e_2/id] \Downarrow v}{(\lambda id.e_1)\, e_2 \Downarrow v}$

(APP) nicht vorhanden

(APP-LEFT) $\dfrac{e_1 \Downarrow v_1 \quad v_1\, e_2 \Downarrow v}{e_1\, e_2 \Downarrow v}$

(APP-RIGHT) $\dfrac{e_2 \Downarrow v_2 \quad v_1\, v_2 \Downarrow v}{v_1\, e_2 \Downarrow v}$ falls v_1 nicht von der Form $\lambda id.e$

herleiten lässt.

3.2.2 Small step Semantik von $\mathcal{L}_0^{CBN}$

Ein small step heißt gültig für $\mathcal{L}_0^{CBN}$, wenn er sich mit den small step Regeln von $\mathcal{L}_0$ und den geänderten bzw. gelöschten Regeln

(BETA-V) nicht vorhanden

(BETA) $(\lambda id.e_1)\, e_2 \rightarrow e_1[e_2/id]$

(APP-RIGHT) $\dfrac{e \rightarrow e'}{v\, e \rightarrow v\, e'}$ falls v nicht von der Form $\lambda id.e_0$

herleiten lässt.

3.3 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:

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

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

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

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

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

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

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

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.

3.3.1 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) $\dfrac{e_1 \Downarrow \mathit{false}}{e_1 \mathbin{\&\&} e_2 \Downarrow \mathit{false}}$

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

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

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

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

(COERCE) $\dfrac{e \Downarrow v}{(e \ : \ \tau \ <: \ \tau') \Downarrow v}$

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.

3.3.2 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)	$\dfrac{e_1 \to e_1'}{e_1 \mathbin{\&\&} e_2 \to e_1' \mathbin{\&\&} e_2}$
(AND-FALSE)	$\mathit{false} \mathbin{\&\&} e_2 \to \mathit{false}$
(AND-TRUE)	$\mathit{true} \mathbin{\&\&} e_2 \to e_2$
(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)	$\mathit{true} \parallel e_2 \to \mathit{true}$
(COERCE)	$(e \ : \ \tau \ <: \ \tau') \to e$

und mit den zugehörigen exception-Regeln herleiten lässt. Diese exceptions-Regeln erhält man - ähnlich wie bei der big step Semantik - 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) $\dfrac{e_1 \rightarrow \mathbf{exn}}{e_2 \rightarrow \mathbf{exn}}$

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

3.3.3 Typechecker Semantik 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. Diese sollten 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) $\dfrac{c :: \tau}{\Gamma \triangleright c :: \tau}$

(ID) $\Gamma \triangleright id :: \tau$ falls $id \in dom(\Gamma)$ und $\Gamma(id) = \tau$

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

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

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

(ABSTR) $\dfrac{\Gamma[\tau/id] \triangleright e :: \tau'}{\Gamma \triangleright \lambda id : \tau . e :: \tau \rightarrow \tau'}$

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

(OR) $\dfrac{\Gamma \triangleright e_1 :: \mathbf{bool} \quad \Gamma \triangleright e_2 :: \mathbf{bool}}{\Gamma \triangleright e_1 \parallel e_2 :: \mathbf{bool}}$

(COERCE) $\dfrac{\Gamma \triangleright e :: \tau \quad \tau <: \tau'}{\Gamma \triangleright (e : \tau <: \tau') :: \tau'}$

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 Type Checker werden diese Regeln für Konstanten nicht angegeben, sondern lediglich die (CONST) Regel.

3.3.4 Minimal Typing Semantik von $\mathcal{L}_1$

Ein Typurteil $\Gamma \rhd_m e :: \tau$ oder $\Gamma \rhd_m r :: \phi$ heißt gültig für $\mathcal{L}_o^m$, wenn es sich mit den Typregeln

(ID)	$\Gamma \rhd_m id :: \tau$ falls $id \in dom(\Gamma) \land \Gamma(id) = \tau$
(CONST)	$\dfrac{c :: \tau}{\Gamma \rhd_m c :: \tau}$
(APP-SUBSUME)	$\dfrac{\Gamma \rhd_m e_1 :: \tau_2' \to \tau \quad \Gamma \rhd e_2 :: \tau_2 \quad \tau_2 <: \tau_2'}{\Gamma \rhd_m e_1\, e_2 :: \tau}$
(ABSTR)	$\dfrac{\Gamma[{}^{\tau}/_{x}] \rhd_m e :: \tau'}{\Gamma \rhd_m \lambda x : \tau.e :: \tau \to \tau'}$
(LET)	$\dfrac{\Gamma \rhd_m e_1 :: \tau_1 \quad \Gamma[{}^{\tau_1}/_{x}] \rhd_m e_2 :: \tau_2}{\Gamma \rhd_m \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 :: \tau_2}$
(COND-SUBSUME)	$\dfrac{\Gamma \rhd_m e_0 :: \mathbf{bool} \quad \Gamma \rhd_m e_1 :: \tau_1 \quad \Gamma \rhd_m e_2 :: \tau_2}{\Gamma \rhd_m \mathbf{if}\, e_0\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2 :: \tau_1 \vee \tau_2}$
(AND)	$\dfrac{\Gamma \rhd_m e_1 :: \mathbf{bool} \quad \Gamma \rhd_m e_2 :: \mathbf{bool}}{\Gamma \rhd_m e_1\, \&\&\, e_2 :: \mathbf{bool}}$
(OR)	$\dfrac{\Gamma \rhd_m e_1 :: \mathbf{bool} \quad \Gamma \rhd_m e_2 :: \mathbf{bool}}{\Gamma \rhd_m e_1 \parallel e_2 :: \mathbf{bool}}$
(COERCE)	$\dfrac{\Gamma \rhd e :: \tau \quad \tau <: \tau'}{\Gamma \rhd (e : \tau <: \tau') :: \tau'}$

und den Suptyping Regeln herleiten läßt.

3.3.5 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

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

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

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

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

Dies sei aber nur am Rande erwähnt.

3.4 Die Sprache $\mathcal{L}_1^{CBN}$

Die Sprache $\mathcal{L}_1^{CBN}$ bietet den gleichen Funktionsumfang wie die Sprache $\mathcal{L}_1$, benutzt aber statt der Call by Value eine Call by Name Semantik. Um dies zu realisieren werden verschiedene Regeln in der big step und der small step Semantik gelöscht oder geändert. Nur diese geänderten bzw. gelöschten Regeln werden hier aufgeführt.

3.4.1 Big step Semantik von $\mathcal{L}_1^{CBN}$

Ein big step heißt gültig für $\mathcal{L}_1^{CBN}$, wenn er sich mit den big step Regeln von $\mathcal{L}_1$ und den geänderten bzw. gelöschten Regeln

(BETA-V) nicht vorhanden

(BETA) $\dfrac{e_1[e_2/id] \Downarrow v}{(\lambda id.e_1)\ e_2 \Downarrow v}$

(APP) nicht vorhanden

(APP-LEFT) $\dfrac{e_1 \Downarrow v_1 \quad v_1\ e_2 \Downarrow v}{e_1\ e_2 \Downarrow v}$

(APP-RIGHT) $\dfrac{e_2 \Downarrow v_2 \quad v_1\ v_2 \Downarrow v}{v_1\ e_2 \Downarrow v}$ falls v_1 nicht von der Form $\lambda id.e$

(LET) $\dfrac{e_2[e_1/id] \Downarrow v}{\mathbf{let}\ id = e_1\ \mathbf{in}\ e_2 \Downarrow v}$

herleiten lässt.

3.4.2 Small step Semantik von $\mathcal{L}_1^{CBN}$

Ein small step heißt gültig für $\mathcal{L}_1^{CBN}$, wenn er sich mit den small step Regeln von $\mathcal{L}_1$ und den geänderten bzw. gelöschten Regeln

(BETA-V) nicht vorhanden

(BETA) $(\lambda id.e_1)\, e_2 \rightarrow e_1[e_2/id]$

(APP-RIGHT) $\dfrac{e \rightarrow e'}{v\, e \rightarrow v\, e'}$ falls v nicht von der Form $\lambda id.e_0$

(LET-EVAL) nicht vorhanden

(LET-EXEC) $\textbf{let}\ id = e_1\ \textbf{in}\ e_2 \rightarrow e_2[e_1/id]$

herleiten lässt.

3.5 Die Sprache $\mathcal{L}_1^{SUB}$

Die Sprache $\mathcal{L}_1^{SUB}$beschränkt sich auf das Typsystem von $\mathcal{L}_1$.

3.5.1 Sub Typing Semantik von $\mathcal{L}_1^{SUB}$

Ein Beweisschritt für Sub Typing heißt gültig, wenn er sich mit den Regeln

(S-REFL) $\tau\ <:\ \tau$

(S-ARROW) $\dfrac{\tau_1'\ <:\ \tau_1 \quad \tau_2\ <:\ \tau_2'}{\tau_1 \rightarrow \tau_2\ <:\ \tau_1' \rightarrow \tau_2'}$

herleiten lässt.

3.5.2 Rec Sub Typing Semantik von $\mathcal{L}_1^{SUB}$

Ein Beweisschritt für Rec Sub Typing heißt gültig, wenn er sich mit den Regeln

(S-REFL) $A \vdash \tau \ <: \ \tau$

(S-ASSUME) $\dfrac{(\tau_1 \ <: \ \tau_2) \in A}{A \vdash \tau_1 \ <: \ \tau_2}$

(S-ARROW) $\dfrac{A' \vdash \tau_1 \ <: \ \tau_1' \quad A' \vdash \tau_2 \ <: \ \tau_2'}{A \vdash \tau_1 \to \tau_2 \ <: \ \tau_1' \to \tau_2'}$ mit $A' = A \cup \{\tau_1 \to \tau_2 \ <: \ \tau_1' \to \tau_2'\}$

(S-MU-LEFT) $\dfrac{A' \vdash \tau_1[^{\mu t.\tau_1}/_t] \ <: \ \tau_2}{A \vdash \mu t.\tau_1 \ <: \ \tau_2}$ mit $A' = A \cup \{\mu t.\tau_1 \ <: \ \tau_2\}$

(S-MU-RIGHT) $\dfrac{A' \vdash \tau_1 \ <: \ \tau_2[^{\mu t.\tau_2}/_t]}{A \vdash \tau_1 \ <: \ \mu t.\tau_2}$ mit $A' = A \cup \{\tau_1 \ <: \ \mu t.\tau_2\}$

herleiten lässt.

3.6 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

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

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.

3.6.1 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

(UNFOLD) $\dfrac{e[\mathbf{rec}\ id.e/id] \Downarrow v}{\mathbf{rec}\ id.e \Downarrow v}$

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.

3.6.2 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.

3.6.3 Typechecker Semantik 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

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

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

3.6.4 Minimal Typing Semantik von $\mathcal{L}_2$

Ein Typurteil $\Gamma \triangleright_m e :: \tau$ oder $\Gamma \triangleright_m r :: \phi$ heißt gültig für $\mathcal{L}_2$, wenn es sich mit den Typregeln

$$\text{(REC-SUBSUME)} \quad \frac{\Gamma[{}^{\tau}/_{x}] \triangleright_m e :: \tau' \quad \tau' \leq \tau}{\Gamma \triangleright_m \textbf{rec}\, x : \tau.e :: \tau}$$

und den Suptyping Regeln herleiten lässt.

3.6.5 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.

3.7 Die Sprache $\mathcal{L}_2^{CBN}$

Die Sprache $\mathcal{L}_2^{CBN}$ bietet den gleichen Funktionsumfang wie die Sprache $\mathcal{L}_2$, benutzt aber statt der Call by Value eine Call by Name Semantik. Um dies zu realisieren werden verschiedene Regeln in der big step und der small step Semantik gelöscht oder geändert. Nur diese geänderten bzw. gelöschten Regeln werden hier aufgeführt.

3.7.1 Big step Semantik von $\mathcal{L}_2^{CBN}$

Ein big step heißt gültig für $\mathcal{L}_2^{CBN}$, wenn er sich mit den big step Regeln von $\mathcal{L}_2$ und den geänderten bzw. gelöschten Regeln

(BETA-V) nicht vorhanden

(BETA) $\dfrac{e_1[e_2/id] \Downarrow v}{(\lambda id.e_1)\, e_2 \Downarrow v}$

(APP) nicht vorhanden

(APP-LEFT) $\dfrac{e_1 \Downarrow v_1 \quad v_1\, e_2 \Downarrow v}{e_1\, e_2 \Downarrow v}$

(APP-RIGHT) $\dfrac{e_2 \Downarrow v_2 \quad v_1\, v_2 \Downarrow v}{v_1\, e_2 \Downarrow v}$ falls v_1 nicht von der Form $\lambda id.e$

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

herleiten lässt.

3.7.2 Small step Semantik von $\mathcal{L}_2^{CBN}$

Ein small step heißt gültig für $\mathcal{L}_2^{CBN}$, wenn er sich mit den small step Regeln von $\mathcal{L}_2$ und den geänderten bzw. gelöschten Regeln

(BETA-V)	nicht vorhanden
(BETA)	$(\lambda id.e_1)\, e_2 \rightarrow e_1[e_2/id]$
(APP-RIGHT)	$\dfrac{e \rightarrow e'}{v\,e \rightarrow v\,e'}$ falls v nicht von der Form $\lambda id.e_0$
(LET-EVAL)	nicht vorhanden
(LET-EXEC)	$\textbf{let}\ id = e_1\ \textbf{in}\ e_2 \rightarrow e_2[e_1/id]$

herleiten lässt.

3.8 Die Sprache $\mathcal{L}_2^{SUB}$

Die Sprache $\mathcal{L}_2^{SUB}$erweitert die Sprache $\mathcal{L}_1$ um das Typsystem der Sprache $\mathcal{L}_2$.

3.9 Die Sprache $\mathcal{L}_2^{O}$

Die Sprache $\mathcal{L}_2^{O}$ erweitert die Sprache $\mathcal{L}_2$ um Objekte. Die Menge *Exp* der gültigen Ausdrücke wird hierfür um die Produktionen

$$
\begin{array}{rcll}
e & ::= & (e\#m) & \text{Methodenaufruf} \\
 & | & \textbf{object}\ (self : \tau)\ r\ \textbf{end} & \text{Objekt} \\
 & | & \{<\ a_1 = e_1;\ \ldots\ ; a_n = e_n\ >\} & \text{Duplikation}
\end{array}
$$

erweitert. Die Menge *Row* aller *Reihen* r von $\mathcal{L}_2^{O}$ ist definiert durch die kontextfreie Grammatik:

$$
\begin{array}{rcll}
r & ::= & \varepsilon & \text{leere Reihe} \\
 & | & \textbf{val}\ a = e;\ r_1 & \text{Attribut} \\
 & | & \textbf{method}\ m : \tau = e\ ;\ r_1 & \text{Methode}
\end{array}
$$

Die Menge $Val_r \subseteq Row$ aller *Reihenwerte* von $\mathcal{L}_2^O$ ist definiert durch die kontextfreie Grammatik:

$$\begin{array}{lcll} \omega & ::= & \varepsilon & \text{leere Reihe} \\ & | & \mathbf{val}\ a = v;\ \omega_1 & \text{Attribut} \\ & | & \mathbf{method}\ m : \tau = e\ ;\ \omega_1 & \text{Methode} \end{array}$$

Die Menge *Val* der Werte v wird um die Produktionen

$$\begin{array}{lcll} v & ::= & \mathbf{object}\ (self : \tau)\ \omega\ \mathbf{end} & \text{Objektwert} \end{array}$$

erweitert, wobei sämtliche Typangaben wieder optional sind.

Die Menge der monomorphen Typen *Type* wird erweitert durch eine Produktion für Objekte

$$\begin{array}{lcl} \tau & ::= & < \phi > \end{array}$$

und die Menge $Type_r$ aller *Reihentypen* ϕ von $\mathcal{L}_2^O$ ist durch

$$\begin{array}{lcl} \phi & ::= & \emptyset \\ & | & m : \tau;\ \phi_1 \end{array}$$

definiert, wobei die Methodennamen in einem Reihentyp ϕ paarweise verschieden sein müssen.

3.9.1 Big step Semantik von $\mathcal{L}_2^O$

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

(OBJECT) $\dfrac{r \Downarrow \omega}{\textbf{object}\ (self)\ r\ \textbf{end} \Downarrow \textbf{object}\ (self)\ \omega\ \textbf{end}}$

(SEND) $\dfrac{e \Downarrow \textbf{object}\ (self)\ \omega\ \textbf{end} \quad \omega[\textbf{object}\ (self)\ \omega\ \textbf{end}/self]\#m \Downarrow v}{e\#m \Downarrow v}$

(SEND-ATTR) $\dfrac{\omega[v/a]\#m \Downarrow v'}{(\textbf{val}\ a = v; \omega)\#m \Downarrow v'}$

(SEND-SKIP) $\dfrac{m \neq m' \vee m \in dom_m(\omega) \quad \omega\#m \Downarrow v}{(\textbf{method}\ m' = e\ ; \omega)\#m \Downarrow v}$

(SEND-EXEC) $\dfrac{m = m' \wedge m \notin dom_m(\omega) \quad e \Downarrow v}{(\textbf{method}\ m' = e\ ; \omega)\#m \Downarrow v}$

(OMEGA) $\omega \Downarrow \omega$

(ATTR) $\dfrac{e \Downarrow v \quad r \Downarrow \omega}{\textbf{val}\ a = e; r \Downarrow \textbf{val}\ a = v; \omega}$

(METHOD) $\dfrac{r \Downarrow \omega}{\textbf{method}\ m = e\ ; r \Downarrow \textbf{method}\ m = e\ ; \omega}$

sowie den dazugehörigen exception-Regeln herleiten lässt.

3.9.2 Small step Semantik von $\mathcal{L}_2^O$

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

(OBJECT-EVAL) $$\frac{r \to r'}{\textbf{object}\ (self)\ r\ \textbf{end} \to \textbf{object}\ (self)\ r'\ \textbf{end}}$$

(SEND-EVAL) $$\frac{e \to e'}{e\#m \to e'\#m}$$

(SEND-UNFOLD) $$\textbf{object}\ (self)\ \omega\ \textbf{end}\#m \to \omega[\textbf{object}\ (self)\ \omega\ \textbf{end}/self]\#m$$

(SEND-ATTR) $$(\textbf{val}\ a = v; \omega)\#m \to \omega[v/a]\#m$$

(SEND-SKIP) $$(\textbf{method}\ m' = e\ ; \omega)\#m \to \omega\#m \quad falls\ m \neq m' \vee m \in dom_m(\omega)$$

(SEND-EXEC) $$(\textbf{method}\ m' = e\ ; \omega)\#m \to e \quad falls\ m = m' \wedge m \notin dom_m(\omega)$$

(ATTR-EVAL) $$\frac{e \to e'}{\textbf{val}\ a = e; r \to \textbf{val}\ a = e'; r}$$

(ATTR-RIGHT) $$\frac{r \to r'}{\textbf{val}\ a = v; r \to \textbf{val}\ a = v; r'}$$

(METHOD-RIGHT) $$\frac{r \to r'}{\textbf{method}\ m = e\ ; r \to \textbf{method}\ m = e\ ; r'}$$

sowie den entsprechenden exception-Regeln herleiten lässt.

3.9.3 Typechecker Semantik von $\mathcal{L}_2^O$

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

(SEND) $$\frac{\Gamma \triangleright e :: < m : \tau; \phi >}{\Gamma \triangleright e\#m :: \tau}$$

(OBJECT) $$\frac{\Gamma^*[\tau/self] \triangleright r :: \phi \quad \tau = < \phi >}{\Gamma \triangleright \textbf{object}\ (self : \tau)\ r\ \textbf{end} :: \tau}$$

(DUPL) $$\frac{\Gamma \triangleright self :: \tau \quad \forall i = 1 \ldots n : \Gamma \triangleright a_i :: \tau_i \wedge \Gamma \triangleright e_i :: \tau_i}{\Gamma \triangleright \{< a_1 = e_1; \ldots; a_n = e_n >\} :: \tau}$$

(EMPTY) $$\Gamma \triangleright \epsilon :: \emptyset$$

(ATTR) $$\frac{\Gamma^* \triangleright e :: \tau \quad \Gamma[\tau/a] \triangleright r_1 :: \phi}{\Gamma \triangleright \textbf{val}\ a = e; r_1 :: \phi}$$

(METHOD) $$\frac{\Gamma \triangleright e :: \tau \quad \Gamma \triangleright r_1 :: \phi}{\Gamma \triangleright \textbf{method}\ m = e\ ; r_1 :: m : \tau; \phi}$$

herleiten lässt. Γ^* ist die Typumgebung mit der Eigenschaft, dass nur Identifier enthalten sind, die nicht zu einem Objekt oder Attribut gehören.

3.9.4 Minimal Typing Semantik von $\mathcal{L}_2^O$

Ein Typurteil $\Gamma \rhd_m e :: \tau$ oder $\Gamma \rhd_m r :: \phi$ heisst gültig für $\mathcal{L}_o^m$, wenn es sich mit den Typregeln

$$\text{(SEND)} \quad \frac{\Gamma \rhd_m e :: \langle m : \tau;\, \phi \rangle}{\Gamma \rhd_m e\#m :: \tau}$$

$$\text{(OBJECT)} \quad \frac{\Gamma^\star[{}^\tau/_{self}] \rhd_m r :: \phi \quad \tau = \langle \phi \rangle}{\Gamma \rhd_m \textbf{object}\,(self : \tau)\; r\; \textbf{end} :: \tau}$$

$$\text{(DUPL-SUBSUME)} \quad \frac{\Gamma \rhd_m self :: \tau \quad \forall i : \Gamma \rhd_m a_i :: \tau_i \wedge \Gamma \rhd_m e_i :: \tau_i' \wedge \tau_i' \leq \tau_i}{\Gamma \rhd_m \{\langle a_1 = e_1; \ldots; a_n = e_n \rangle\} :: \tau}$$

$$\text{(EMPTY)} \quad \Gamma \rhd_m \epsilon :: \emptyset$$

$$\text{(ATTR)} \quad \frac{\Gamma^\star \rhd_m e :: \tau \quad \Gamma[{}^\tau/_a] \rhd_m r_1 :: \phi}{\Gamma \rhd_m \textbf{val}\; a = e\; ;\; r_1 :: \phi}$$

$$\text{(METHOD-SUBSUME)} \quad \frac{\Gamma \rhd_m self :: \langle m : \tau;\, \phi' \rangle \quad \Gamma \rhd_m e :: \tau' \quad \Gamma \rhd_m r_1 :: \phi \quad \tau' \leq \tau}{\Gamma \rhd_m \textbf{method}\; m = e\; ;\; r_1 :: (m : \tau;\, \emptyset) \oplus \phi}$$

und den Suptyping Regeln herleiten lässt.

3.9.5 Syntaktischer Zucker

Die Sprache $\mathcal{L}_2^O$ enthält weitere Abkürzungen zu den in $\mathcal{L}_2$ definierten. Die folgende Abkürzung steht für die leichtere Definition von Funktionen zur Verfügung.

$$\textbf{method}\; m(id_1 : \tau_1) \ldots (id_n : \tau_n) : \tau = e\; ;$$

steht für

$$\textbf{method}\; m : \tau = \lambda id_1 : \tau_1. \ldots \lambda id_n : \tau_n.e\; ;$$

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

3.10 Die Sprache $\mathcal{L}_2^{OSUB}$

Die Sprache $\mathcal{L}_2^{OSUB}$erweitert die Sprache $\mathcal{L}_2^{SUB}$um das Typsystem von $\mathcal{L}_2^{O}$.

3.10.1 Sub Typing Semantik von $\mathcal{L}_2^{OSUB}$

Ein Beweisschritt für Sub Typing heißt gültig, wenn er sich mit den Regeln

(S-TRANS) $$\frac{\tau_1 \ <: \ \tau_2 \quad \tau_2 \ <: \ \tau_3}{\tau_1 \ <: \ \tau_3}$$

(S-OBJ-WIDTH) $$\langle m_1 : \tau_1; \ldots; m_{n+k} : \tau_{n+k} \rangle \ <: \ \langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle$$

(S-OBJ-DEPTH) $$\frac{\tau_i \ <: \ \tau_i' \text{ für } i = 1, \ldots, n}{\langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle \ <: \ \langle m_1 : \tau_1'; \ldots; m_n : \tau_n' \rangle}$$

(S-OBJECT) $$\frac{\tau_i \ <: \ {}_m\tau_j' \text{ für alle } i, j \text{ mit } m_i = m_j'}{\langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle \ <: \ {}_m\langle m_1' : \tau_1'; \ldots; m_l' : \tau_l' \rangle}$$

falls $\{m_1', \ldots, m_l'\} \subseteq \{m_1, \ldots, m_k\}$

herleiten lässt.

3.10.2 Rec Sub Typing Semantik von $\mathcal{L}_2^{OSUB}$

Ein Beweisschritt für Rec Sub Typing heißt gültig, wenn er sich mit den Regeln

(S-TRANS) $$\frac{A \vdash \tau_1 \ <: \ \tau_2 \quad A \vdash \tau_2 \ <: \ \tau_3}{A \vdash \tau_1 \ <: \ \tau_3}$$

(S-OBJ-WIDTH) $$A \vdash \langle m_1 : \tau_1; \ldots; m_{n+k} : \tau_{n+k} \rangle \ <: \ \langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle$$

(S-OBJ-DEPTH) $$\frac{A' \vdash \tau_i \ <: \ \tau_i' \text{ für } i = 1, \ldots, n}{A \vdash \langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle \ <: \ \langle m_1 : \tau_1'; \ldots; m_n : \tau_n' \rangle}$$

mit $A' = A \cup \{\langle m_1 : \tau_1; \ldots; m_n : \tau_n \rangle \ <: \ \langle m_1 : \tau_1'; \ldots; m_n : \tau_n' \rangle\}$

(S-OBJECT) $$\frac{A' \vdash \tau_i \ <: \ {}_m\tau_j' \text{ für alle } i, j \text{ mit } m_i = m_j'}{\langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle \ <: \ {}_m\langle m_1' : \tau_1'; \ldots; m_l' : \tau_l' \rangle}$$

mit $A' = A \cup \{\langle m_1 : \tau_1; \ldots; m_k : \tau_k \rangle \ <: \ {}_m\langle m_1' : \tau_1'; \ldots; m_l' : \tau_l' \rangle\}$

falls $\{m_1', \ldots, m_l'\} \subseteq \{m_1, \ldots, m_k\}$

herleiten lässt.

3.11 Die Sprache $\mathcal{L}_2^C$

Die Sprache $\mathcal{L}_2^C$ erweitert die Sprache $\mathcal{L}_2^O$ um Klassen bzw. Vererbung. Die Menge *Exp* der gültigen Ausdrücke wird hierfür um die Produktionen

$$\begin{array}{rcll} e & ::= & \textbf{class}\ (self : \tau)\ b\ \textbf{end} & \text{Klassen} \\ & | & \textbf{new}\ e & \text{New} \end{array}$$

erweitert.

Die Menge *Body* aller *Klassenrümpfe* b von $\mathcal{L}_2^C$ ist definiert durch die kontextfreie Grammatik:

$$\begin{array}{rcll} b & ::= & \textbf{inherit}\ a_1,\ \ldots\ ,a_k\ \textbf{from}\ e\ ;\ b & \text{Inherit} \\ & | & r & \text{Reihe} \end{array}$$

Die Menge *Val* der Werte v wird um die Produktionen

$$\begin{array}{rcll} v & ::= & \textbf{class}\ (self : \tau)\ r\ \textbf{end} & \text{Klassenwert} \end{array}$$

erweitert, wobei sämtliche Typangaben wieder optional sind.

Die Menge der monomorphen Typen *Type* wird erweitert durch eine Produktion für Klassen

$$\tau \ ::= \ \zeta(\tau : \phi)$$

und die Menge Type_r aller *Reihentypen* ϕ von $\mathcal{L}_2^C$ wird erweitert durch

$$\phi \ ::= \ a : \tau;\ \phi_1$$

wobei die Methodennamen und Attributnamen in einem Reihentyp ϕ paarweise verschieden sein müssen.

3.11.1 Big step Semantik von $\mathcal{L}_2^C$

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

(CLASS) $$\frac{b \Downarrow r}{\textbf{class}\ (self)\ b\ \textbf{end} \Downarrow \textbf{class}\ (self)\ r\ \textbf{end}}$$

(NEW) $$\frac{e \Downarrow \textbf{class}\ (self)\ r\ \textbf{end}}{\textbf{new}\ e \Downarrow \textbf{object}\ (self)\ r\ \textbf{end}}$$

(INHERIT) $$\frac{b \Downarrow r_2 \quad e \Downarrow \textbf{class}\ (self)\ r_1\ \textbf{end} \quad dom_a\,(r_1) = A}{\textbf{inherit}\ A\ \textbf{from}\ e\ ;\ b \Downarrow r_1 \oplus r_2}$$

sowie den dazugehörigen exception-Regeln herleiten lässt.

Bei dem Symbol $\oplus$ handelt es sich um die Vereinigung der beiden Reihen r_1 und r_2. Das Ergebniss ist eine Reihe, in der erst die Kinder der Reihe r_1 vorkommen und danach die Kinder der Reihe r_2.

Die Bedingung $dom_a\,(r_1) = A$ bedeutet, dass alle angegebenen Attributnamen in dem Inherit Teil mit den Attributen in der Reihe r_1 übereinstimmen müssen. Ist dies nicht der Fall, bleibt der Small Step Interpreter stecken.

3.11.2 Small step Semantik von $\mathcal{L}_2^C$

Ein small step heißt gültig für $\mathcal{L}_2^C$, wenn er sich mit den small step Regeln von $\mathcal{L}_2^O$ und den Regeln

(CLASS-EVAL) $$\frac{b \to b'}{\textbf{class}\ (self)\ b\ \textbf{end} \to \textbf{class}\ (self)\ b'\ \textbf{end}}$$

(NEW-EVAL) $$\frac{e \to e'}{\textbf{new}\ e \to \textbf{new}\ e'}$$

(NEW-EXEC) $$\textbf{new}\ (\textbf{class}\ (self)\ r\ \textbf{end}) \to \textbf{object}\ (self)\ r\ \textbf{end}$$

(INHERIT-RIGHT) $$\frac{b \to b'}{\textbf{inherit}\ A\ \textbf{from}\ e\ ;\ b \to \textbf{inherit}\ A\ \textbf{from}\ e\ ;\ b'}$$

(INHERIT-LEFT) $$\frac{e \to e'}{\textbf{inherit}\ A\ \textbf{from}\ e\ ;\ r \to \textbf{inherit}\ A\ \textbf{from}\ e'\ ;\ r}$$

(INHERIT-EXEC) $$\textbf{inherit}\ A\ \textbf{from class}\ (self)\ r_1\ \textbf{end}\ ;\ r_2 \to r_1 \oplus r_2$$ falls $dom_a\,(r_1) = A$

sowie den entsprechenden exception-Regeln herleiten lässt.

Bei dem Symbol $\oplus$ handelt es sich um die Vereinigung der beiden Reihen r_1 und r_2. Das Ergebniss ist eine Reihe, in der erst die Kinder der Reihe r_1

vorkommen und danach die Kinder der Reihe r_2.

Die Bedingung $dom_a\,(r_1) = A$ bedeutet, dass alle angegebenen Attributnamen in dem Inherit Teil mit den Attributen in der Reihe r_1 übereinstimmen müssen. Ist dies nicht der Fall, bleibt der Small Step Interpreter stecken.

3.12 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

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

und

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

erweitert. Für Listenoperationen wird eine weitere Ausnahme

$$\textbf{exn} \ ::= \ empty_list$$

hinzugenommen.

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

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

3.12.1 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)	$\dfrac{e_1 \Downarrow v_1 \quad \dots \quad e_n \Downarrow v_n}{(e_1, \dots, e_n) \Downarrow (v_1, \dots, v_n)}$
(FST)	$\mathit{fst}\,(v_1, v_2) \Downarrow v_1$
(SND)	$\mathit{snd}\,(v_1, v_2) \Downarrow v_2$
(PROJ)	$\#n_i\,(v_1, \dots, v_n) \Downarrow v_i \qquad (1 \leq i \leq n)$
(LIST)	$\dfrac{e_1 \Downarrow v_1 \quad \dots \quad e_n \Downarrow v_n}{[e_1; \dots; e_n] \Downarrow [v_1; \dots; v_n]}$
(CONS)	$(::)\,v'\,[v_1; \dots; v_n] \Downarrow [v'; v_1; \dots; v_n]$
(HD)	$\mathit{hd}\ (\mathit{cons}\ (v_1, v_2)) \Downarrow v_1$
(HD-EMPTY)	$\mathit{hd}\,[\,] \Downarrow\uparrow \mathit{empty_list}$
(TL)	$\mathit{tl}\ (\mathit{cons}\ (v_1, v_2)) \Downarrow v_2$
(TL-EMPTY)	$\mathit{tl}\,[\,] \Downarrow\uparrow \mathit{empty_list}$
(IS-EMPTY-FALSE)	$\mathit{is_empty}\ (\mathit{cons}\,v) \Downarrow \mathit{false}$
(IS-EMPTY-TRUE)	$\mathit{is_empty}\,[\,] \Downarrow \mathit{true}$

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 $(\mathit{cons}\,v) \in \mathit{Val}$. Für Details zum Thema Listen sei hier auf den Inhalt der Vorlesung verwiesen.

3.12.2 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

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

von $\mathcal{L}_2$ und den Regeln

(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)	$\mathit{fst}\,(v_1, v_2) \to v_1$
(SND)	$\mathit{snd}\,(v_1, v_2) \to v_2$
(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)	$\mathit{hd}\ (\mathit{cons}\ (v_1, v_2)) \to v_1$
(HD-EMPTY)	$\mathit{hd}\,[\,] \to\uparrow \mathit{empty_list}$
(TL)	$\mathit{tl}\ (\mathit{cons}\ (v_1, v_2)) \to v_2$
(TL-EMPTY)	$\mathit{tl}\,[\,] \to\uparrow \mathit{empty_list}$
(IS-EMPTY-FALSE)	$\mathit{is_empty}\ (\mathit{cons}\, v) \to \mathit{false}$
(IS-EMPTY-TRUE)	$\mathit{is_empty}\,[\,] \to \mathit{true}$

sowie den entsprechenden exception-Regeln herleiten lässt.

3.12.3 Typechecker Semantik 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

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

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

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

die bisherige Regel (ID). Die neue Regel

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

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

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

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

hinzugenommen.

3.12.4 Minimal Typing Semantik von $\mathcal{L}_3$

Ein Typurteil $\Gamma \triangleright_m e :: \tau$ oder $\Gamma \triangleright_m r :: \phi$ heisst gültig für $\mathcal{L}_3$, wenn es sich mit den Typregeln

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

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

und den Suptyping Regeln herleiten lässt.

3.12.5 Syntaktischer Zucker

$\mathcal{L}_3$ enthält syntaktischen Zucker für den leichteren Umgang mit Listen und Tupeln. Die Akürzungen werden 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**

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

steht für

$$\textbf{let } id : \tau_1 * \ldots * \tau_n = e_1 \textbf{ in let } id_1 = (\#n_1\, id) \textbf{ in} \ldots \textbf{let } id_n = (\#n_n\, id) \textbf{ in } e_2$$

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.

3.13 Die Sprache $\mathcal{L}_3^{SUB}$

Die Sprache $\mathcal{L}_2^{SUB}$erweitert die Sprache $\mathcal{L}_2^{SUB}$um das Typsystem von $\mathcal{L}_3$.

3.13.1 Sub Typing Semantik von $\mathcal{L}_3^{SUB}$

Ein Beweisschritt für Sub Typing heißt gültig, wenn er sich mit den Regeln

(S-LIST) $$\frac{\tau \;<:\; \tau'}{\tau\,\textbf{list} \;<:\; \tau'\,\textbf{list}}$$

(S-PRODUCT) $$\frac{\tau_i \;<:\; \tau_i \text{ für i = 1 ... n}}{\tau_1 * \ldots * \tau_n \;<:\; \tau_1' * \ldots * \tau_n'}$$

herleiten lässt.

3.13.2 Rec Sub Typing Semantik von $\mathcal{L}_3^{SUB}$

Ein Beweisschritt für Rec Sub Typing heißt gültig, wenn er sich mit den Regeln

(S-LIST) $$\frac{A' \vdash \tau \;<:\; \tau'}{A \vdash \tau\,\textbf{list} \;<:\; \tau'\,\textbf{list}} \quad \text{mit } A' = A \cup \{\tau\,\textbf{list} \;<:\; \tau'\,\textbf{list}\}$$

(S-PRODUCT) $$\frac{A' \vdash \tau_i \;<:\; \tau_i \text{ für i = 1 ... n}}{\tau_1 * \ldots * \tau_n \;<:\; \tau_1' * \ldots * \tau_n'} \quad \text{mit } A' = A \cup \{\tau_1 * \ldots * \tau_n \;<:\; \tau_1' * \ldots * \tau_n'\}$$

herleiten lässt.

3.14 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

$$c \ ::= \ ref \mid ! \mid :=$$

erweitert.

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

$$\sigma : Loc \hookrightarrow Val$$

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

$$alloc : Store \to Loc$$

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

3.14.1 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)	$(ref\ v, \sigma) \Downarrow (X, \sigma[v/X])$ mit $X = alloc(\sigma)$
(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)	$\dfrac{(e_0, \sigma) \Downarrow (false, \sigma')}{(\textbf{if}\ e_0\ \textbf{then}\ e_1, \sigma) \Downarrow ((), \sigma')}$
(COND-1-TRUE)	$\dfrac{(e_0, \sigma) \Downarrow (true, \sigma') \quad (e_1, \sigma') \Downarrow (v_1, \sigma'')}{(\textbf{if}\ e_0\ \textbf{then}\ e_1, \sigma) \Downarrow (v_1, \sigma'')}$
(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.

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

Ein small step ist nun von der Form $(e, \sigma) \to (e', \sigma')$ oder $(e, \sigma) \to (exn, \sigma)$. 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)	$(ref\ v, \sigma) \to (X, \sigma[v/X])$ mit $X = alloc(\sigma)$
(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)	$(\textbf{if}\ false\ \textbf{then}\ e, \sigma) \to ((), \sigma)$
(COND-1-TRUE)	$(\textbf{if}\ true\ \textbf{then}\ e, \sigma) \to (e, \sigma)$
(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.

3.14.3 Typechecker Semantik von $\mathcal{L}_4$

Ein Typurteil heißt gültig für $\mathcal{L}_4$, wenn es sich mit den Typregeln von $\mathcal{L}_3$ und den Regeln

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

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

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

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

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

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

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

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.

3.14.4 Syntaktischer Zucker

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

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

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

3.15 Die Sprache $\mathcal{L}_4^{SUB}$

Die Sprache $\mathcal{L}_4^{SUB}$erweitert die Sprache $\mathcal{L}_3^{SUB}$um das Typsystem von $\mathcal{L}_4$.

3.15.1 Sub Typing Semantik von $\mathcal{L}_4^{SUB}$

Ein Beweisschritt für Sub Typing heißt gültig, wenn er sich mit den Regeln

$$\text{(S-REF)} \quad \frac{\tau \ <: \ \tau'}{\tau\,\mathbf{ref} \ <: \ \tau'\,\mathbf{ref}}$$

herleiten lässt.

3.15.2 Rec Sub Typing Semantik von $\mathcal{L}_4^{SUB}$

Ein Beweisschritt für Rec Sub Typing heißt gültig, wenn er sich mit den Regeln

$$\text{(S-REF)} \quad \frac{A' \vdash \tau \ <: \ \tau'}{A \vdash \tau\,\mathbf{ref} \ <: \ \tau'\,\mathbf{ref}} \quad \text{mit } A' = A \cup \{\tau\,\mathbf{ref} \ <: \ \tau'\,\mathbf{ref}\}$$

herleiten lässt.

3.16 Konkrete Syntax

Da es mitunter nicht mit jeder herkömmlichen Tastatur möglich ist, griechische Buchstaben zu tippen, können die aufgeführten Schlüsselwörter als Ersatz benutzt werden. Ebenso ist es nicht so einfach Unicode Zeichen einzugeben, auch hierfür gibt es die entsprechenden Abkürzungen. Unterscheidungen, die nur in der Theorie möglich sind, in der Praxis aber nicht unterscheidbar sind, müssen mit Schlüsselwörtern unterschieden werden.

λ	**lambda**	Lambda Ausdruck
μ	**mu**	Rekursive Typen
ζ	**zeta**	Klassentypen
$\alpha,\ \beta,\ \ldots$	`'a`, `'b`, ...	Typvariablen
$\rightarrow$	`->`	Funktionstypen
$a:\tau_1;\ m:\tau_2;\ \phi$	**attr** $a:\tau_1;\ m:\tau_2;\ \phi$	Reihentypen

Bei den Reihentypen muss eine Entscheidung getroffen werden, ob es sich bei einem eingegebenen Identifier um einen Attributnamen oder um einen Methodennamen handelt. Da der Parser das nicht automatisch erkennen kann, muss das Schlüsselwort „`attr`“ verwendet werden, wenn der Benutzer ein Attributnamen meint. Wird das Schlüsselwort nicht eingegeben, handelt es sich um einen Methodennamen.

Zu beachten ist auch, dass der Scanner den Ausdruck „$(*)\,4\,5$“ nicht akzeptiert, da Kommentare mit einem „$(*$“ eingeleitet werden. Er erwartet ein „$*)$“, um den Kommentar zu beenden. Wenn multipliziert werden soll, muss „$(\ *)\,4\,5$“ eingeben werden.

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 beseitigt 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, dass man die Regel verstanden hat.

Anhang A

Lizenz

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


```
Copyright (c) 2005-2007 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.
```


Der von uns verwendete **CUP Parser Generator** steht unter folgender Lizenz:

CUP Parser Generator Copyright Notice, License, and Disclaimer
Copyright 1996-1999 by Scott Hudson, Frank Flannery, C. Scott Ananian

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that both
the copyright notice and this permission notice and warranty disclaimer
appear in supporting documentation, and that the names of the authors or
their employers not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.

The authors and their employers disclaim all warranties with regard to this
software, including all implied warranties of merchantability and fitness.
In no event shall the authors or their employers be liable for any special,
indirect or consequential damages or any damages whatsoever resulting from loss
of use, data or profits, whether in an action of contract, negligence or other
tortious action, arising out of or in connection with the use or performance
of this software.

This is an open source license. It is also GPL-Compatible
(see entry for "Standard ML of New Jersey").
The portions of CUP output which are hard-coded into the CUP source code
are (naturally) covered by this same license,
as is the CUP runtime code linked with the generated parser.

Java is a trademark of Sun Microsystems, Inc. References to the Java programming
language in relation to JLex are not meant to imply that Sun endorses this product.

Das von uns verwendete **Log4J** steht unter folgender Lizenz:

Copyright 1999-2005 The Apache Software Foundation.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,

```
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```


Das von uns verwendete **itext** steht unter folgender Lizenz:


```
This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Library General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.Â  See the GNU
Library General Public License for more details.

You should have received a copy of the GNU Library General Public
License along with this library; if not, write to the
Free Software Foundation, Inc., 59 Temple Place - Suite 330,
Boston, MA 02111-1307, USA.
```


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

Const, 26, 46
Exp, 35, 38, 46
Loc, 52
Op, 27
Row, 38
Type, 26, 39, 46
$Type_r$, 39, 44
Val, 24, 27, 39, 46
Val_r, 39
λ-Abstraktion, 24
v, 24
Body, 44
Exp, 44
Type, 44
Val, 44

Applikation, 24
Ausnahme, 26
Auto-Vervollständigung, 10

big step, 24, 25, 27, 33, 35, 37, 39, 44, 47, 52

Call By Name, 25, 33, 37
constant, 26

exceptions, 26

imperative Konzepte, 52
Infixschreibweise, 32

Kernsyntax, 32, 37, 42, 54
Klassen, 44
Konfiguration, 52
Konstante, 26

Listen, 46
Lizenz, 58–60
locations, 52

Objekte, 38
Operator, 27
operator, 27
Outline, 11

polymorphes Typsystem, 48
Polymorphie, 46
pure untyped λ-calculus, 23

reiner ungetypter λ-Kalkul, 23
Rekursion, 35

Schleifen, 52
sequentielle Ausfuhrung, 52
small step, 24, 25, 29, 33, 36, 38, 40, 45, 47, 53
Speicher, 52
Speicherplatz, 52
Speicherzustand, 52
store, 52
syntaktischer Zucker, 32

Tupel, 46
Typ, 26
type, 26
type environment, 30
Typinferenzalgorithmus, 30
Typsicherheit, 54

Typsystem, 30, 46
Typumgebung, 30
Typurteil, 30, 36, 41, 54

value, 24
Vererbung, 44

Wert, 24