ESSO – Extensible Shell with Scheme Orientation

Matthias Köppe*

4. Mai 1998

Zusammenfassung

ESSO ist die Shell des Studentischen Betriebssystems (StuBS), welches im Sommersemester 1997 und Wintersemester 1997/98 an der Universität Magdeburg entwickelt wurde; eine Linux-Version von ESSO ist verfügbar. Im Gegensatz zu herkömmlichen Unix-Shells ist ESSO eine (fast) vollwertige Implementation einer gebräuchlichen Programmiersprache, nämlich des Lisp-Dialekts Scheme. Dadurch hat ESSO auch eine besonders einfache Syntax, welche fast ohne Sonderzeichen auskommt. Durch das Konzept der „externen Variablen“ entsteht die Shell-Funktionalität: Ein Verzeichnis-Name evaluiert zu einer Liste mit dem Verzeichnisinhalt; ein Programm-Name zu einer Prozedur, die bei Aufruf einen Programm-Prozeß erstellt; ein Textdatei-Name zu einem „Promise“, die Zeilen der Textdatei zu liefern.

1 ESSO und ihre Scheme-Implementation

Es wurde so ziemlich alles implementiert, was im *Revised^4 Report for the Algorithmic Language Scheme* beschrieben ist. In der aktuellen Version ist ESSO sogar *properly tail-recursive*, das heißt, endrekursive Lambdas erzeugen keine rekursiven Prozesse, so daß nur konstanter Platz benötigt wird. Der *garbage collector* von ESSO ist aber *purely reference-counting*, d. h., es gibt Probleme mit zirkulären Referenzen. An Zahlentypen gibt es nur (kleine) Integers, und Vektoren werden nicht unterstützt.

Auf C++-Ebene sind nur die wichtigsten Sprachelemente implementiert, nämlich die folgenden *special forms*: define, set!, quote, if, and, or, begin, lambda, eval. Über die Variable syntax-alist (eine Assoc-Liste) können eigene Special-Form-Evaluatoren installiert werden. In der Systembibliothek `essolib.scm` werden so die folgenden *special forms* definiert: let, let*, delay, cond, case, sequence, quasiquote. Quasiquotations arbeiten bislang nur unvollständig.

Ebenfalls ist auf C++-Ebene nur ein „minimaler“ Satz von primitiven Prozeduren implementiert; der Rest an Funktionalität wird über Lambdas aus der Systembibliothek bereitgestellt. Einige auf Scheme-Ebene realisierbare Prozeduren wurden aber aus Geschwindigkeitsgründen als Primitive in C++ implementiert.

In ESSO Scheme werden „Promises“ (etwa mit delay erzeugt) implizit aufgelöst; die Anwendung von force ist nicht erforderlich.

*`mkoeppe@cs.uni-magdeburg.de`

Findet ESSO in der aktuellen Frame-Hierarchie eine angeforderte Variable nicht, so wird die Funktion eval-external-symbol aufgerufen. Diese ist in der Systembibliothek definiert.

2 Info-System

Schreibt ein verwirrter Nutzer help oder ? (natürlich ohne Listenklammern), so wird eine kurze Hilfemeldung ausgegeben.

Es sind Prozeduren zur Untersuchung des aktuellen Variablen-Frame definiert. (bindings) liefert eine Liste aller gebundenen Variablen. (procedures) liefert eine Liste aller gebundenen Variablen, die Prozeduren speichern; auf -aux endende Hilfsprozeduren werden dabei weggelassen. (variables) liefert eine Liste aller gebundenen Variablen, die keine Prozeduren speichern.

Ein einfaches Hilfesystem wurde implementiert. Es wird mit (info 'symbol) aufgerufen und liefert Informationen zum angegebenen Symbol. Die Informationstexte kommen aus `.sci`-Dateien.

3 Shell-Funktionalität

Auf Scheme-Ebene wurde ein vollständiges Directory-System implementiert. Es gibt das Konzept eines aktuellen Verzeichnisses; Funktionen chdir (cd), pwd sind implementiert. Relative Pfade werden unterstützt.

Eine search-directory-list und search-extension-list enthalten die Suchpfade und Sucherweiterungen. Sie werden in der Funktion search-path verwendet.

Externe Variablen werden (in eval-external-symbol) wie folgt aus dem Dateisystem geladen: Es wird mittels search-path versucht, die benannte Datei (vervollständigt um Erweiterung und Verzeichnis) im Dateisystem zu finden. Es wird nun die file-type-alist, welche Extensions auf Filetypen abbildet, verwendet. Bekannte Filetypen sind: directory, scheme, executable, scheme-data, text. Ihnen entsprechend werden unterschiedliche Aktionen zum Laden der Datei ausgeführt. Daraus ergibt sich die folgende Funktionalität:

1. Der Name eines Verzeichnisses evaluiert zu einer Liste der Namen der im Verzeichnis enthaltenen Dateien.

2. Der Name einer Textdatei evaluiert zu einem Promise, eine Liste der Zeilen der Textdatei zu liefern.

3. Der Name einer ausführbaren Datei evaluiert zu einem Lambda, welches bei Anwendung auf eine Argumentliste das Programm starten wird; Details werden im nächsten Abschnitt beschrieben.

Zur leichteren Lesbarkeit langer Listen empfiehlt es sich, die Funktion pretty einzusetzen. Insbesondere können Textdateien mittels (pretty foo.txt) gelesen werden.

ESSO besitzt einen sehr einfachen Kommandozeileneditor. Er zeigt Klammerstrukturen (*'electric parens'*) und ermöglicht Vervollständigung der Eingabe mit der Tab-Taste. Die eigentliche Vervollständigung wird dabei durch das Lambda tab bewirkt.

4 Kommunikation mit Programmen

Die willkürliche Unterteilung von Programm-Eingaben in Kommandozeilenparameter und Standard-Eingabe gibt es in StuBS/ESSO nicht. Jedes Programm verhält sich wie eine Funktion: Es erhält eine Liste von Argumenten und liefert eine Ausgabe. Die Kommunikation zwischen ESSO und den Programm-Prozessen erfolgt textbasiert über Streams (bzw. `char*`-Pakete, die in eine Mailbox gelegt werden).

Das Starten eines Prozesses für ein Programm übernimmt das Primitiv run-executable; es liefert den Standardeingabe- und Standardausgabeport des Prozesses. Der Name einer ausführbaren Datei evaluiert (mittels load-executable) zu einem Lambda, welches bei Anwendung auf eine Argumentliste die folgenden Aktionen ausführt:

1. Mittels run-executable wird der Prozeß für das Programm erzeugt.
2. Mittels task-force wird ein neuer ESSO-Prozeß erzeugt, welcher die Argumentliste in der durch write-to-executable bestimmten Textrepräsentation an die Standardeingabe des Programmprozesses schreibt.
3. Es wird ein Promise zurückgeliefert, Zeilen aus der Standardausgabe des Programmprozesses zu liefern.

Auf diese Weise werden potentiell unbeschränkte Eingaben und Ausgaben unterstützt. Dies ermöglicht das Ausführen von interaktiven Programmen:

```
(foo 4711 'bar (interactive))
```

Das heißt: Das Programm foo erhält die angegebenen Argumente sowie die aktuelle Standardeingabe, aus der es nach Belieben lesen kann.

In früheren Implementationen wurde zur Definition von interactive die folgende Funktion verwendet:

```
(define (lazy-load-text port)
  (let ((line (read-line port)))
    (if (eof-object? line)
        '()
        (cons line (delay (lazy-load-text port))))))
```

Die zeilenweise Bereitstellung der Standardeingabe bereitete jedoch Probleme mit interaktiven Programmen, welche zeichenweise lesen und schreiben. Deshalb wurde auf C++-Ebene eine spezielle Promise-Klasse implementiert, welche mittels der Primitive make-port-read-promise und handle-port-read-promise angesprochen wird. *Port-read promises* verhalten sich wie das obige lazy-load-text, können aber auch die Daten ihres Ports zeichenweise an einen anderen Port kopieren.

Um in der Version von ESSO für Linux-basierte GNU-Systeme ein Programm mit Unix-Kommandozeilenargumenten auszuführen, ist die folgende Form zu verwenden:

```
((load-executable "/bin/ls -l"))
```

5 Aspekte der Nebenläufigkeit

Mittels des Primitivs task-force können verschiedene nebenläufige ESSO-Prozesse erzeugt werden, die auf dem gleichen Variablenraum arbeiten. Es stellt sich die Frage, wieviel Aufwand zur Synchronisation hier betrieben werden muß.

Scheme-Objekte werden erst dann zerstört, wenn ESSO beweisen kann, daß keine Referenz auf sie mehr existiert. Deshalb ist es nicht möglich, daß ein Prozeß Objekte zerstört, die für einen anderen noch von Bedeutung sind.

Solange keine Mutatoren (set!, set-car!, set-cdr!, string-set!) verwendet werden, werden zudem aus Scheme-Sicht niemals Werte verändert. Deshalb wäre in diesem Fall keinerlei Synchronisation zwischen den nebenläufigen Prozessen notwendig.

Aus C++-Sicht gibt es darüberhinaus eine einzige Struktur, die sich verändern kann, obwohl kein Mutator angewandt wurde: Die `TLispFrame::Map`, implementiert durch einen Rot-Schwarz-Baum, wird durch die `TLispFrame::Define`-Operation geändert. Es muß verhindert werden, daß ein Zugriff auf einen `TLispFrame` erfolgt, der gerade verändert wird; das ist mittels einer Semaphore (je `TLispFrame`-Objekt) realisierbar.

Fernerhin ist das vom *garbage collector* verwendete *reference counting* anfällig gegenüber Nebenläufigkeit. Setzen wir jedoch voraus, daß die Operatoren `++(int &)` und `--(int &)` unteilbar sind,[1] so werden die für das Referenzzählen verantwortlichen Funktionen `TLispObject::Increase()` und `Decrease()` korrekt arbeiten; denn selbst bei einer Überlappung zweier `Decrease()`-Aufrufe erfolgt das Zerstören (`TLispObject::DeleteThis()`) genau beim Nulldurchgang des Referenzzählers.

Die Verwendung der Mutatoren set!, set-car!, set-cdr! führt auf einen Aufruf von `TLispRef::operator=(const TLispRef &)`. Dieser ist so geschrieben, daß – wiederum Unteilbarkeit von `operator=(*&, const *&)` vorausgesetzt – in einer `TLispRef` niemals ein ungültiger Zeiger gespeichert ist. Dadurch können lesende Zugriffe jederzeit erfolgen; es ist keine Absicherung für sie erforderlich. Die Überlappung zweier solcher Zuweisungen auf die gleiche `TLispRef` muß jedoch vermieden werden; dies kann jedoch nur bei Zuweisungen auf Lisp-Variablen (set!) oder die Elemente eines `TLispPair` (set-car!, set-cdr!) geschehen. Deshalb müssen Semaphoren für die Funktionen `LispSetCar()`, `LispSetCdr()` sowie für den set!-Zweig von `lisp_eval()` eingerichtet werden. Der Mutator string-set! muß nicht abgesichert werden (er verändert ein einzelnes Zeichen eines Strings).

Der parallele Zugriff verschiedener Prozesse auf den gleichen Port ist nicht sinnvoll. Falls überhaupt benötigt, kann eine Synchronisation auf Scheme-Ebene erfolgen.

6 Technische Details

ESSO verwendet standardmäßig nicht mehr das schwergewichtige C++-Exception-Handling, sondern eine eigene Exception-Behandlung. Dies verkleinert die Codegröße und beschleunigt die Abarbeitung erheblich.

Soll weiterhin das C++-Exception-Handling verwendet werden, so ist zu beachten: Wegen eines Fehlers in gcc 2.7.2.1 muß die `.s`-(Assembler-)Datei nach-

[1] Das ist im Target i386 (gcc-compiled) der Fall. In Umgebungen ohne diese Eigenschaft müßten `TLispObject::Increase()` und `Decrease()` abgesichert werden.

bearbeitet werden, wenn mit Optimierung compiliert wird. Dazu liegen Skripte bei, die vom `Makefile` verwendet werden.

Da ESSO sehr viele kleine dynamische Datenstrukturen verwendet (99% sind höchstens 16 Byte groß), ist ein sehr effizienter Speichermanager erforderlich. Auf den Betriebssystem-Speichermanager wurde deshalb ein eigener feingranularer Manager aufgesetzt.

Die Prozeßsynchronisation und die Mailbox-basierte Kommunikation von StuBS werden durch C++-Klassen, nämlich `TPipeBase<char>`-Nachfolger, gekapselt; diese Objekte werden auf Scheme-Ebene als Ports angesprochen. Die Prozesse selbst haben auf Scheme-Ebene keine Repräsentation; sie sind nur durch die mit ihnen assoziierten Eingabe-/Ausgabeports zugänglich.

7 Beispiele

Aktuelles Verzeichnis

```
esso@localhost:/ > (pwd)
====> "/"
esso@localhost:/ > (cd "/home/mkoeppe/bs/s/esso")
====> "/home/mkoeppe/bs/s/esso"
```

Rechnen in ESSO

```
esso@localhost:/home/mkoeppe/bs/s/esso > (define (fak n)
  (if (< n 2) n (* n (fak (- n 1)))))
====> fak
esso@localhost:/home/mkoeppe/bs/s/esso > (fak 6)
====> 720
```

Externe Variablen

```
esso@localhost:/home/mkoeppe/bs/s/esso > ./
====> ("." ".." "lisp.h" "Makefile" "lisp.cc" "stpipe.h" "esso.cc"
"essolib.scm" "esso" "filt.sh" "fix-s.sh" "bpipe.h" "ext-test.scm"
"buf-test.scm")
esso@localhost:/home/mkoeppe/bs/s/esso > (car lisp.h)
====> "// This is a -*- C++ -*- header file."
```

Gepufferte externe Lambdas

```
esso@localhost:/home/mkoeppe/bs/s/esso > (cat "buf-test.scm")
====> (define buf-test (lambda (n) (* n n)))
buf-test
esso@localhost:/home/mkoeppe/bs/s/esso > buf-test
====> [a lambda]
esso@localhost:/home/mkoeppe/bs/s/esso > (buf-test 7)
====> 49
```

Infosystem

```
esso@moose:/home/mkoeppe/bs/s/esso > (info)
Welcome to the Info system of Esso.

Type (info 'VARIABLENAME) or (info 'PROCEDURENAME) to get
information about the named variable or procedure. You can
get a list of defined procedures by typing (procedures),
and a list of defined variables by typing (variables).
====> #t
esso@moose:/home/mkoeppe/bs/s/esso > (procedures)
====> (* + - < <= = > >= abs add-to-search-directory-list
... ... ... write-line write-to-executable zero?)
esso@moose:/home/mkoeppe/bs/s/esso > (info 'expand-path)
(expand-path STRING) returns the given path name in an
absolute and canonicalized form.
====> #t
```

Relative Pfade

```
esso@localhost:/home/mkoeppe/bs/s/esso > (cd "../.././s")
====> "/home/mkoeppe/bs/s"
esso@localhost:/home/mkoeppe/bs/s > (cd "thiswontexist")
====> #f
```

Ausnahmebehandlung

```
esso@localhost:/home/mkoeppe/bs/s > (+ 1 "abc")
XInvalidType: string (expected number)
```

Beenden von ESSO

```
esso@localhost:/home/mkoeppe/bs/s/esso > (exit)
====> "Have a nice day."
```