Eine rationale Dekonstruktion von Landin's SECD-Maschine

Olivier Danvy
BRICS_, Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
(danvy@brics.dk)

Betreuerin : Prof. Dr. Rita Loogen
Bearbeiter : Dong Liang
Datum: 12.07.05

Inhaltsverzeichnis

1. Einführung der SECD Maschine..3
1.1 Einleitung..3
1.2 Struktur der SECD Maschine..3
1.3 Vorbedingungen und Gebiet der Darlegung..................................3

2. Dekonstruktion der SECD Maschine..5
2.1 Die originale SECD Maschine..5
2.2 Eine strukturiertere Spezifikation..8
2.3 Höhere Ordnung...10
2.4 Ohne Dump-Fortsetzung...11
2.5 Ohne Control-Fortsetzung...13
2.6 Ohne Stack 14

3. Fazit...16
3.1 Zusammenfassung der Dekonstruktion der SECD Maschine.......16
3.2 Ziel der SECD Maschine..17

4. Referenz...17

1 Einführung der SECD Maschine

1.1 Einleitung

Die SECD-Maschine Landins war die erste abstrakte Maschine für den λ-Kalkül, der als Programmiersprache angesehen wurde. Sie ist ein theoretische Modell für Berechnungen des λ-Kalküls. Die SECD-Maschine ist die Basis für die weitere Entwicklung von abstrakten Maschinen für funktionale Programmiersprache. In diesem Artikel dekonstruieren wir die SECD-Maschine in einen λ-Interpreter d.h. in eine Auswertungsfunktion. Umgekehrt kann aus dem λ-Interpreter die SECD-Maschine wieder aufgebaut werden. Die Dekonstruktion und die Rekonstruktion erfolgen durch Transformation, Gleichungsumformungen, einfache Programmtransformationen, hautsächlich closure Konversion, Transformation in Fortsetzungsart und Defunktionalisierung.

1.2 Struktur der SECD Maschine

Die SECD Maschine wird als eine Übergangsfunktion über einem vierfachen Tupel definiert, bestehend aus einem Stack für Zwischenwerte (mit Typ S), einer Umgebung (mit Typ E), einem Control-stack (mit Typ C) und einem Dump (mit Typ D):

```
run : S * E * C * D -> value
```

Diese Übergangsfunktion ist schwierig, weil sie einige Induktionsvariablen beinhaltet. Deswegen betrachten wir es zum ersten in vier Übergangsfunktionen, von denen jede eine Induktionsvariable hat,

```
run_c :   S * E * C * D -> value
run_d :   S * D -> value
run_t :   term * S * E * C * D -> value
run_a :   S * E * C * D -> value
```

Abhängig von dem Control-stack, ruft run_c entweder run_d auf, (der Control-stack ist leer), oder run_t (Die Spitze des Control-stack enthält einen Term) oder run_a (der Topf des Control-stack enthält eine Applikation).

1.3 Vorbedingungen und Gebiet der Darlegung

Die SECD-Machine ist eine abstrakte Maschine zur Berechnung von Ausdrücken des λ-kalküls. Um die SECD-Machine zu verstehen, müssen wir zunächst den λ-kalkül bzw. die Ausgangssprache (*The source language*)

kennenlernen.

The source language : Die Ausgangssprache ist der λ-Kalkül. Ein Programm ist ein geschlossene Term. Er wird wie folgt definiert:

```
structure Source
= struct
        type ide = string
        datatype term = LIT of int
                      | VAR of ide
                      | LAM of ide * term
                      | APP of term * term
        type program = term
end
```

Ein Term ist eine integer Konstante int(c), einer Variablen ide(x), eine λ-Abstraktion LAM ide*term(λx.e), oder ein Apply term*term(e,e'). Wir können auch wie folgt schreiben:

c :: =c | x | λx.e | (e e')

Und danach definieren wir die Umgebung (*The (polymorphic) environment)*
Wir definieren eine Struktur Env, welche die folgende Signatur hat:

```
signature ENV
= sig
      type 'a env
      val empty : 'a env
      val extend : Source.ide * 'a * 'a env -> 'a env
      val lookup : Source.ide * 'a env -> 'a
  end
```

Auf der Umgebung ENV sind drei Operationen definiert. Die leere Umgebung wird mit Env.empty bezeichnet. Die Funktion, die eine Umgebung mit einer neuen Bindung erweitert, wird mit Env.extend bezeichnet. Die Funktion, die den Wert eines Bezeichners aus einer Umgebung holt, wird mit Env.lookup bezeichnet.

Am Ende definieren wir die Wertebreichtyp (value). Es gibt drei Typen von Werten: Integer, die Nachfolgefunktion, und die closure:

```
datatype value = INT of int
               | SUCC
               | CLOSURE of value Env.env * Source.ide * Source.term
```

Mit den Definitionen der Quellsprache, der Umgebung, und der Werte können wir jetzt die initiale Umgebung *(The initial environment)* definieren:

```
val e_init = Env.extend ("succ", SUCC, Env.empty)
```

2. Dekonstruktion der SECD Maschine

Um die SECD Maschine genauer verstehen und analysieren zu können, dekonstruieren wir die Maschine. Am Ende bauen wir sie wieder auf.
Die Betrachtung gliedert sich in 6 Abschnitten:

Abschnitt 2.1 stellt die SECD-Maschine dar, wie sie ursprünglich spezifiziert und in der Literatur klassisch dargestellt wird.

Abschnitt 2.2 beinhaltet eine alternative Spezifikation, in der die Funktion run in run_c, run_d, run_t und run_a mit vier gegenseitig rekursiven Übergangsfunktionen transformiert wird, von denen jede eine Induktionsvariable besitzt.

Abschnitt 2.3 stellt eine Version höherer Ordnung dar. Diese arbeitet mit Fortsetzungsfunktionen (continuation-passing-style).

In Abschnitt 2.4 wird die Dump-Fortsetzung eliminiert.

In Abschnitt 2.5 wird noch die Control-Fortsetzung eliminiert.

Abschnitt 2.6 stellt den entsprechenden Experten ohne Stack dar.

2.1 Die originale SECD Maschine

Die SECD Maschine wird mit Übergängen zwischen vier Komponenten definiert:

-- **S**tack_register enthält eine Liste von Zwischenresultaten. Diese Komponente hat den Typ **value list.**

-- **E**nvironment_register enthält eine Liste von Umgebungen. Diese ist vom Typ **value Env.env.**

-- **C**ontrol_register enthält Control directives. Diese Komponente hat den Typ **Directive,** wobei Directive wie folgt definiert ist:

```
datatype directive = TERM of Source.term
                   | APPLY
```

-- **D**ump_ register enthält eine Liste von Dreiergruppen. Die Dreiergruppe enthält die Stackliste, die Umgebung, und den Control Register. Die Komponeten sind vom Typ **(value list * value Env.env * directive list) list.**

Die SECD Maschine ist mit der Transition zwischen vier Komponeten definiert:

```
(*   run : S * E * C * D -> value     *)
(*   where S = value list             *)
(*         E = value Env.env          *)
(*         C = directive list         *)
(*         D = (S * E * C) list       *)

(* evaluate0 : Source.program -> value *)
 fun evaluate0   t
      = run (nil, e_init, (TERM t) :: nil, nil)
```

Die Auswertungsfunktion wird mit einem leeren Stack, der Umgebung, dem initiale Ausdruck und einem leeren Dump initialisiert.

Die originale SECD Maschine

```
fun run (v :: nil, e', nil, nil)                                    (* 1 *)
      = v
  | run (v :: nil, e', nil, (s, e, c) :: d)                         (* 2 *)
      = run (v :: s, e, c, d)
  | run (s, e, (TERM (LIT n)) :: c, d)                              (* 3 *)
      = run ((INT n) :: s, e, c, d)
  | run (s, e, (TERM (VAR x)) :: c, d)                              (* 4 *)
      = run ((Env.lookup (x, e)) :: s, e, c, d)
  | run (s, e, (TERM (LAM (x, t))) :: c, d)                         (* 5 *)
      = run ((CLOSURE (e, x, t)) :: s, e, c, d)
  | run (s, e, (TERM (APP (t0, t1))) :: c, d)                       (* 6 *)
      = run (s, e, (TERM t1) :: (TERM t0) :: APPLY :: c, d)
  | run (SUCC :: (INT n) :: s, e, APPLY :: c, d)                    (* 7 *)
      = run ((INT (n+1)) :: s, e, c, d)
  | run ((CLOSURE (e', x, t)) :: v' :: s, e, APPLY :: c, d)         (* 8 *)
      = run (nil, Env.extend (x, v', e'), (TERM t) :: nil, (s, e, c) :: d)
```

1. Die erste Klausel spezifiziert, wenn die Liste der Control und das Dump leer sind, die Berechnung wird beendet und der Wert wird auf der Spitze des Stack zurückgegeben.

2. Die zweite Klausel spezifiziert, wenn die Liste der Control leer aber das Dump nicht leer ist, die Berechnung sollte mit dem Stack, der Umgebung und der Control fortfahren, die in der Spitze des Dumps gespeichert wird und den obersten Wert des Stacks auf den neuen Stack bringen.

3. Die dritte Klausel spezifiziert, wenn die Spitze der Controlliste eine Integer Zahl ist, der Wert sollte auf den Stack eingeführt werden.

4. Die 4. Klausel spezifiziert, wenn die Spitze der Controlliste ein Variable ist, der entsprechende Wert sollte von der entsprechenden Umgebung geholt werden und auf den Stack eingefügt werden.

5. Die 5. Klausel spezifiziert, wenn die Spitze der Controlliste eine λ-Abstraktion ist, die dem Auswerten dieser λ-Abstraktion entspricht: die entsprechende closure sollte auf den Stack eingefügt werden. Dieses closure gruppiert die Umgebung und die zwei Bestandteile der λ-Abstraktion d.h. die Formelparameter und den Körper.

6. Die 6. Klausel spezifiziert, wenn die Spitze der Controlliste eine Anwendung ist, die dem Auswerten einer Anwendung entspricht: eine Applikation, der Operator und die Operanden sollten auf die Liste der Control eingefügt werden.

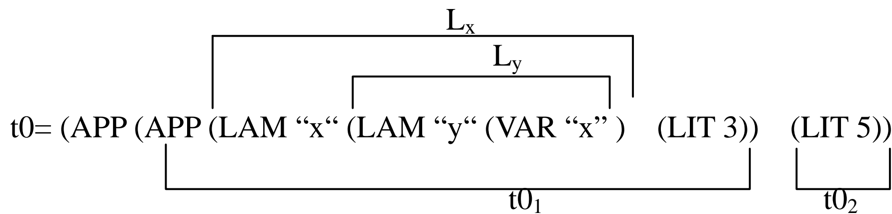
7. Die 7. Klausel spezifiziert, wenn die Spitze der Controlliste ein Apply ist, bei der die Oberseite des Stacks die Nachfolgerfunktion ist und das folgende Controlelement eine Ganze Zahl ist , die der Anwendung der Nachfolgerfunktion entspricht: der Control-Direktor sollte zweimal geknallt werden und die Ganze Zahl sollte auf den Stack erhöht und eingefügt werden.

8. Die achte Klausel spezifiziert, wenn die Spitze der Controlliste ein Apply ist, bei der die Oberseite des Stacks eine closure ist und ein folgendes Element im Stack existiert, das einem Funktionsaufruf entspricht. In diesem Fall sollte der Stack zweimal geknallt werden und die Umgebung sowie der Rest der Liste des Controls auf den Dump eingefügt werden. Der Zwischenwert

Stack sollte mit der leeren Liste initialisiert werden, die Umgebung sollte mit dem closure Umgebung initialisiert werden.

Ein Beispiel für die originale SECD Maschine

Bsp1: ((λx.λy.x   3)   5) mit evaluate0



= run (nil, e_init, (TERM t0) : : nil, nil)
(6) = run (nil, e_init, (TERM $t0_2$) : : (TERM $t0_1$) : : APPLY : : nil, nil)
(3) = run ((INT 5), e_init, (TERM $t0_1$) : : APPLY : : nil, nil)
(6) = run ((INT 5), e_init, (LIT 3) : : (TERM L_x) : : APPLY : : APPLY : : nil, nil)
(3) = run ((INT 3) : : (INT 5), e_init, (TERM L_x) : : APPLY : : APPLY : : nil, nil)
(5) = run ((CLOSURE (e_init, “x”, L_y) : : (INT 3) : : (INT 5), e_init,
APPLY : : APPLY : : nil, nil)
(8) = run (nil, [(“x”, (INT 3))], (TERM L_y) : : nil, ((INT 5), e_init, APPLY) : : nil)
(5) = run ((CLOSURE([(“x”, (INT 3))], “y”, VAR”x”) : : nil, [(“x”, (INT 3))],
[ ],((INT 5), e_init, APPLY) : : nil)
(2) = run ((CLOSURE([(“x”, (INT 3))], “y”, VAR”x”) : : (INT 5) : : nil, e_init,
APPLY, nil)
(8) = run (nil, [(“x”, (INT 3)), (“y”, (INT 5))], (TERM VAR”x”) : : nil,
(nil, e_init, nil) : : nil)
(4) = run ((INT 3) : : nil, [(“x”, (INT 3)), (“y”, (INT 5))], nil, (nil, e_init, nil) : : nil)
(2) = run ((INT 3) : : nil, e_init, nil, nil)
(1) = INT 3

Der Term t0 wird mit den Klauseln 1-8 berechnet. Nach 13 Schritte wurde das Resultat INT 3 ausgegeben.

2.2 Eine strukturiertere Spezifikation

In der Definition von Abschnitt 2.1 werden alle möglichen Übergänge zusammen in einer rekursiven Funktion **run** definiert.

Jetzt wandelt Faktor **run** in einige gegenseitig rekursive Funktionen um, davon jede mit einer Induktionsvariable.

-- run_c interpretiert die Liste von control directives. Es spezifiziert, welche Transition zu nehmen ist. Wenn die Liste leer ist, wird run_d aufgerufen. Wenn die Liste mit einem Term beginnt, erfolgt der Aufruf von run_d. Beginnt die Liste mit einer Anwendung, wird run_a aufgerufen.

-- run_d interpretiert den Dump. Es spezifiziert, welche Transition zu nehmen ist. Falls das Dump leer oder nicht leer ist, gibt es einen gültigen Stack.

-- run_t interpretiert den ersten Term in der Liste von Control.

-- run_a interpretiert das erste Resultat im Stack.

Die strukturiertere Spezifikation wird wie folgt definiert:

```
(*    run_c : S * E * C * D -> value                    *)
(*    run_d : S * D -> value                            *)
(*    run_t : Source.term * S * E * C * D -> value *)
(*    run_a : S * E * C * D -> value                    *)
(*      where S = value list                            *)
(*            E = value Env.env                         *)
(*            C = directive list                        *)
(*            D = (S * E * C) list                      *)

(* evaluate1 : Source.program -> value                  *)
  fun evaluate1    t
    = run_t (t, nil, e_init, nil, nil)
```

Die strukturiertere Spezifikation

```
fun run_c (s, e, nil, d)
        = run_d (s, d)
   | run_c (s, e, (TERM t) :: c, d)
        = run_t (t, s, e, c, d)
   | run_c (s, e, APPLY :: c, d)
        = run_a (s, e, c, d)
and run_d (v :: nil, nil)
        = v
   | run_d (v :: nil, (s, e, c) :: d)
        = run_c (v :: s, e, c, d)
and run_t (LIT n, s, e, c, d)
        = run_c ((INT n) :: s, e, c, d)
   | run_t (VAR x, s, e, c, d)
        = run_c ((Env.lookup (x, e)) :: s, e, c, d)
   | run_t (LAM (x, t), s, e, c, d)
        = run_c ((CLOSURE (e, x, t)) :: s, e, c, d)
   | run_t (APP (t0, t1), s, e, c, d)
        = run_t (t1, s, e, (TERM t0) :: APPLY :: c, d)
```

```
and run_a (SUCC :: (INT n) :: s, e, c, d)
        = run_c ((INT (n+1)) :: s, e, c, d)
   | run_a ((CLOSURE (e', x, t)) :: v' :: s, e, c, d)
        = run_t (t, nil, Env.extend (x, v', e'), nil, (s, e, c) :: d)
```

Vergleich der originalen Maschine und der strukturierteren Spezifikation

Vergleicht man die Definition der strukturierteren Spezifikation mit der originalen Maschine, so lässt sich feststellen, dass die 1. und 2. Klausel der originalen Maschine mit dem run_d der Definition der strukturierteren Spezifikation übereinstimmt. Des Weiteren entsprechen die 3,4,5,6. Klausel run_t und die 7. und 8. Klausel entsprechen run_a. Damit folgt die erste Proposition:

Proposition 1: Die Quellprogramme evaluate0 und evaluate1 liefern das gleiche Resultat.

2.3 Höhere Ordnung

In Abschnitt 2.2, gibt es zwei Möglichkeiten um einen Dump zu konstruieren (Nil und Cons) und drei Möglichkeiten um eine Liste von Control zu konstruieren (nil, a term, and apply directive).
Betrachtet man es in höherer Ordnung, so definieren run_d und run_c Funktionen in Form von Fortsetzungsfunktionen.

```
(*   run_t : Source.term * S * E * C * D -> value    *)
(*   run_a : S * E * C * D -> value                  *)
(*        where S = value list                       *)
(*              E = value Env.env                    *)
(*              C = (S * E * D) -> value             *)
(*              D = S -> value                       *)

(* evaluate2 : Source.program -> value *)
fun evaluate2 t
     = run_t (t, nil, e_init,
              fn (s, _, d) => d s,
              fn (v :: nil) => v)
```

Hier arbeitet evaluate2 mit Fortsetzungsfunktionen (continuation-
-passing-style). Die zwei Fortsetzungsfunktionen **c** und **d** haben die Gestalt:

```
C = (S * E * D) -> value
D = S -> value
```

Die höhere Ordnung ist wie folgt definiert:

```
fun run_t (LIT n, s, e, c, d)
        = c ((INT n) :: s, e, d)
  | run_t (VAR x, s, e, c, d)
        = c ((Env.lookup (x, e)) :: s, e, d)
  | run_t (LAM (x, t), s, e, c, d)
        = c ((CLOSURE (e, x, t)) :: s, e, d)
  | run_t (APP (t0, t1), s, e, c, d)
        = run_t (t1, s, e,
                 fn (s, e, d) => run_t (t0, s, e,
                 fn (s, e, d) => run_a (s, e, c, d),
                 d),
          d)
and run_a (SUCC :: (INT n) :: s, e, c, d)
        = c ((INT (n+1)) :: s, e, d)
  | run_a ((CLOSURE (e', x, t)) :: v' :: s, e, c, d)
        = run_t (t, nil, Env.extend (x, v', e'),
                 fn (s, _, d) => d s,
                 fn (v :: nil) => c (v :: s, e, d))
```

Vergleich mit Strukturierterer Spezifikation

In der Strukturierteren Spezifikation werden run_d und run_c als Transitionsfunktion definiert. Wobei in C die Liste von Control directives gespeichert wird und in D die Liste von (S * E * C) gespeichert ist. Ausserdem werden in höherer Ordnung die C und D als Fortsetzungsfunktionen definiert. C ist die Fortsetzung der (S * E * C) Liste und D wird als die Fortsetzung des Stack interpretiert. Die beiden Strukturen spielen die gleiche Rolle. Es folgt die zweite Proposition:

Proposition 2: Die Quellprogramme evaluate1 und evaluate2 erbringen das gleiche Resultat.

2.4 Ohne dump-Fortsetzung

Die Version von Abschnitt 2.3 arbeitet mit Fortsetzungsfunktionen. In diesem Abschnitt wird die dump-Fortsetzungsfunktion eliminiert.
Wir benennen run_t als eval und run_a als apply um.

```
(*   eval : Source.term * S * E * C -> stack    *)
```

```
(*    apply : S * E * C -> S                         *)
(*        where S = value list                        *)
(*                  E = value Env.env                 *)
(*                  C = S * E -> S                    *)

(* evaluate3 : Source.program -> value *)
  fun evaluate3 t
      = let val (v :: nil) = eval (t, nil, e_init, fn (s, _) => s)
          in v
          end

fun eval (LIT n, s, e, c)
        = c ((INT n) :: s, e)
    | eval (VAR x, s, e, c)
        = c ((Env.lookup (x, e)) :: s, e)
    | eval (LAM (x, t), s, e, c)
        = c ((CLOSURE (e, x, t)) :: s, e)
    | eval (APP (t0, t1), s, e, c)
        = eval (t1, s, e, fn (s, e) =>
            eval (t0, s, e, fn (s, e) =>
            apply (s, e, c)))
and apply (SUCC :: (INT n) :: s, e, c)
        = c ((INT (n+1)) :: s, e)
   | apply ((CLOSURE (e', x, t)) :: v' :: s, e, c)
        = let val (v :: nil) = eval (t, nil, Env.extend (x, v', e'),
                                      fn (s, _) => s)
            in c (v :: s, e)
            end
```

Vergleich mit höherer Ordnung

In diesem Abschnitt wird evaluate mit let val(v :: nil)=...in v definiert. Sie gibt das Resultat direkte in v aus. Die Funktion ist gleich wie die Dump-Fortsetzung (fn(v :: nil) => v), die in höherer Ordnung implementiert ist. eval(APP(t0,t1), s, e, c) arbeitet mit zwei Control-Fortsetzung und rechnet das APPLY mit apply(s, e, c) ohne Dump-Fortsetzungsfunktion. Es spielt die gleiche Rolle wie run_t(APP(t0,t1), s, e, c, d), die in höherer Ordnung definiert ist. Der andere eval und apply Evaluetor ergeben das gleiche Resultat wie run_t und run_a in höherer Ordung. Daraus folgt die Proposition 3:

Proposition 3: Die Quellprogramme evaluate2 und evaluate3 liefern das gleiche Resultat.

2.5 Ohne Control-Fortsetzung

In Abschintt 2.4 wurde die Dump Fortsetzung eliminiert. Nun eliminiern wir noch die Control Fortsetzung:

```
(*      eval : Source.term * S * E -> S * E    *)
(*    apply : S * E -> S * E                   *)
(*         where S = value list                *)
(*                 E = value Env.env           *)

(* evaluate4 : Source.program -> value *)
   fun evaluate4 t
        = let val (v :: nil, _)
                        = eval (t, nil, e_init))
            in v
            end

  fun eval (LIT n, s, e)
          = ((INT n) :: s, e)
     | eval (VAR x, s, e)
          = ((Env.lookup (x, e)) :: s, e)
     | eval (LAM (x, t), s, e)
          = ((CLOSURE (e, x, t)) :: s, e)
     | eval (APP (t0, t1), s, e)
          = let val (s, e) = eval (t1, s, e)
                val (s, e) = eval (t0, s, e)
            in apply (s, e)
            end
and apply (SUCC :: (INT n) :: s, e)
          = ((INT (n+1)) :: s, e)
    | apply ((CLOSURE (e', x, t)) :: v' :: s, e)
          = let val (v :: nil, _)
                        = eval (t, nil, Env.extend (x, v', e')))
             in (v :: s, e)
             end
```

Vergleichen mit Ohne Dump-Fortsetzungsfunktion

In diesem Abschnitt wird der Term direkt ausgerechnet. Er arbeitet nicht mehr mit der Control-Fortsetzung. eval(APP(t0,t1), s, e) berechnet t0 und t1 und speichert die beiden Resultate im Stack, und rechnet die Enderesultate mit apply(s, e). In dem Fall ohne Dump-Fortsetzung arbeitet eval(APP(t0,t1), s, e, c) mit Control-Fortsetzungsfunktion. Es liefert das gleiche Resultat wie ohne Control-Fortsetzung. Auch der eval und apply evaluetor, die in beiden Fälle auftreten, liefern gleiche Resultate.

Proposition 4 : Die Quellprogramme evaluate3 und evaluate4 erbringen das gleiche Resultat.

2.6 Ohne-Stack

Im Abschnitt 2.5 arbeiten eval und apply mit einem Stack von Zwischenresultaten. Wir stellen hier eine Möglichkeit dar, die Resultate direkt auszugeben, ohne Verwendung eines Stacks für die Zwischenwerte.

```
(* eval : Source.term * E -> value * E         *)
(* apply : value * value * E -> value * E    *)
(*       where E = value Env.env                  *)

(* evaluate5 : Source.program -> value *)
   fun evaluate5 t
      = let val (v', _)
               = eval (t, e_init))
         in v'
         end


   fun eval (LIT n, e)
            = (INT n, e)
      | eval (VAR x, e)
            = (Env.lookup (x, e), e)
      | eval (LAM (x, t), e)
            = (CLOSURE (e, x, t), e)
      | eval (APP (t0, t1), e)
            = let val (v1, e) = eval (t1, e)
                  val (v0, e) = eval (t0, e)
               in apply (v0, v1, e)
               end
  and apply (SUCC, INT n, e)
            = (INT (n+1), e)
     | apply (CLOSURE (e', x, t), v', e)
            = let val (v, _)
                     = eval (t, Env.extend (x, v', e')))
               in (v, e)
               end
```

Vergleich mit ohne Control-Fortsetzung

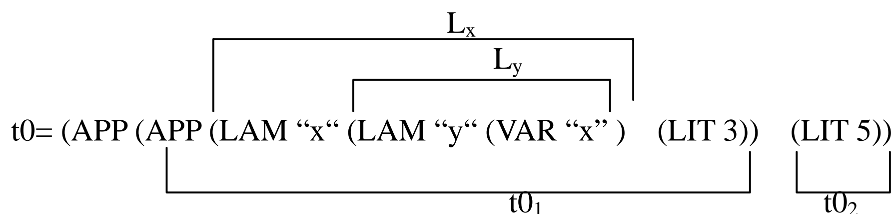
In diesem Abschnitt werden die Zwischenresultate und Endresultate direkt ausgegeben ohne sie in einem Stack zu speichen. Z.B. berechnet <u>eval(APP(t0,t1), e)</u> t0 und t1. Die Resultate werden als v1 und v0 direkt

ausgegeben und dann mit apply(v0,v1,e) weiter gerechnet. Im Fall der Ohne Control-Fortsetzung eval(APP(t0,t1),s,e) werden die Zwischenresultate zunächst im Stack gespeichert und dann das Enderesultat weiter mit apply(s,e) ausgerechnet. Die andere eval und apply evaluetor bleiben gleich.

Proposition 5 : Die Quellprogramme evaluate4 und evaluate5 erbringen das gleiche Resultat.

Ein Beispiel für Abschnitt 2.6 Ohne-Stack

Bsp2: ((λx.λy.x 3) 5) mit evaluate5




```
   = let val ( v', _ )
         = eval ( t0, e_init )
       in v'
       end

eval( t0, e_init )
   = let val ( v1, e1 ) = eval ( t0₂ , e_init )
         val ( v0, e0 ) = eval ( t0₁ , e_init )
     in apply ( v0, v1, e )
     end

eval ( t0₂ , e_init ) = ( LIT 5, e_init ) = ( INT 5, e_init )

eval ( t0₁ , e_init ) = let val ( v3, e3 ) = eval ( LIT 3, e_init ) = ( INT 3, e_init )
                            val ( v2, e2 ) = eval ( Lx , e_init )
                                           = ( CLOSURE (e_init, "x", Ly ), e_init )
                        in apply ( v2, v3, e_init )
                        end

apply ( v2, v3, e_init ) = let val ( v, _ )
                                 = eval ( Ly ,Env.extend ("x", INT 3, e_init ) )
                           in ( v, e_init )
                           end

eval ( Ly ,Env.extend ("x", INT 3, e_init ) )
   = ( CLOSURE ( [ "x", ( INT 3 ) ], "y", VAR"x" ), [ ] )      ( * e_init = [ ] * )
```

➡ eval (t0, e_init)
 = let val (v, _) = eval (VAR"x", [("x", (INT 3)), ("y", INT 5)])
 = (INT 3, [("x", (INT 3)), ("y", INT 5)])
 in (v, [ ])
 end
 = INT 3

Im Beispiel 2 wird der gleiche Term wie im Beispiel 1 berechnet und gleiches Resultat ausgegeben.

3. Fazit

3.1 Zusammenfassung der Dekonstruktion der SECD Machine

Wie in Abschnitt 2.1-2.6 wird die SCED-Maschine Schritt für Schritt dekonstruiert und wieder aufgebaut. Die originale SECD-Maschine wird zur strukturierteren SECD Machine mit vier Transitionsfunktionen spezifiziert.

run_c : S * E * C * D -> value
run_d : S * D -> value
run_t : term * S * E * C * D -> value
run_a : S * E * C * D -> value

Danach werden die beiden Transitionsfunktionen run_c und run _d als Fortsetzung definiert mit Hilfe von Fortsetzungsfunktionen. (continuation-passing style)

run_t : term * S * E * C * D -> value
run_a : S * E * C * D -> value
where C = S * E * D -> value
D = S -> value

Danach wird die Dump-Fortsetzung elimiert:

run_t : term * S * E * C -> stack
run_a : S * E * C -> S
where C = S * E -> S

Und dann elimiert man noch die Control-Fortsetzung.

run_t : S * E -> S * E
run_a : S * E -> S * E

Am Ende elimiert das Stack.

run_t : term * E -> value * E
run_a : value * value * E -> value * E

Nach der Dekonstruktion der SECD Maschine kann man feststellen, dass alle Abschnitte das gleiche Resultat liefern.

3.2 Ziel der SECD Maschine

Das Enderesultat der Dekonstruktion der SECD Maschine zeigt, daß der denotationale Inhalt der Auswertungsfunktion der SECD-Maschine von folgendem Typ ist:

term -> E -> value * E

wobei term ist vom Typ **term ,** E ist vom Typ **Umgebung** und value ist vom Typ **value.**
Mit der oben gezeigten Dekonstruktion und Rekonstruktion der SECD Machine, sollte man verstehen können, wie die SECD Machine der λ-kalkül berechnet.
Es sollte eigentlich der term t bezüglich der Umgebung E direkt das Enderesultat ergeben.

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19. ACM Press, August 2003.
2. Mads Sig Ager, Olivier Danvy, and Mayer Goldberg. A symmetric approach to compilation and decompilation. In Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, number 2566 in Lecture Notes in Computer Science, pages 296–331. Springer-Verlag, 2002.
3. Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the technical report BRICS-RS-04-03.
4. Dariusz Biernacki and Olivier Danvy. From interpreter to logic engine by defunctionalization. Technical Report BRICS RS-03-25, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 2003. To appear in the proceedings of the 2003 International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003).
5. Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.
6. Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.
7. Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report BRICS RS-03-33, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 2003.
8. Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor,

Proceedings of the 1990 ACM Conference on Lisp and Functional Programming, pages 151–160, Nice, France, June 1990. ACM Press.
9. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.
10. Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In Harald Søndergaard, editor, *Proceedings of the Third International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'01)*, pages 162–174, Firenze, Italy, September 2001. ACM Press. Extended version available as the technical report BRICS RS-01-23.
11. Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.
12. Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.
13. Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and Peter Mager, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.
14. Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Unpublished lecture notes. http://www.ccs.neu.edu/home/matthias/3810-w02/readings.html, 1989-2003.
15. Andrzej Filinski. Representing monads. In Boehm [5], pages 446–457.
16. Piet Hein. *Grooks*. The MIT Press, 1966.
17. Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In Olin Shivers, editor, *Proceedings of the 2003 ACM SIGPLAN International Conference on Functional Programming*, pages 177–188, Uppsala, Sweden, August 2003. ACM Press.
18. Jean-Louis Krivine. Un interpr`ete du λ-calcul. Brouillon. Available online at http://www.pps.jussieu.fr/~krivine, 1985.
19. Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
20. Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.
21. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Massachusetts, 1962.
22. Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993.
23. Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
24. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
25. John D. Ramsdell. The tail-recursive SECD machine. *Journal of Automated Reasoning*, 23(1):43–62, July 1999.
26. John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.
27. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
28. Guy L. Steele Jr. Lambda, the ultimate declarative. AI Memo 379, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, November 1976.
29. Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence

Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.
30. Guy L. Steele Jr. and Gerald J. Sussman. Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1976.
31. Guy L. Steele Jr. and Gerald J. Sussman. The art of the interpreter or, the modularity complex (parts zero, one, and two). AI Memo 453, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.
32. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.
33. Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.