Geschachtelte Scopes

```
int i;

void level_0(void) {
    int j;

    void level_1(void) {
        int k;

        void level_2(void) {
            int l;

            ...             /* code has access to i, j, k, l */
            k = l;
            j = l;
        }

        ...                 /* code has access to i, j, k */
        j = k;
    }

    ...                     /* code has access to i, j */
}
```

Figure 6.25 Nested routines in C notation.

Sprung aus einer inneren Routine

```
void level_0(void) {
    void level_1(void) {
        void level_2(void) {
            ...
            goto L_1;
            ...
        }
        ...
    L_1:...
        ...
    }
    ...
}
```

Figure 6.26 Example of a non-local goto.

Implementierung:

- Abräumen aller Aktivierungssegmente von `level_2` seit dem `level_2` von `level_1` aufgerufen wurde
- Wiederherstellen der Registerinhalte vor dem ersten Aufruf von `level_2`
- Fortsetzung des Kontrollflusses bei `L_1`

Rücksprung aus einer dynamischen Folge von Routinenaktivierungen (1)

- Anwendungen
  - schnelles Beenden einer rekursiven Suche
  - Implementierung von Exceptions
- Mechanismus
  - Speicherung einer zur Laufzeit bestimmten Sprungmarke (Gabelung des Kontrollflusses) in einer Variablen
  - Verfolgung des ersten Kontrollflusszweiges
  - Rücksprung
  - Fortsetzung mit dem zweiten Kontrollflusszweig

Rücksprung aus einer dynamischen Folge von Routinenaktivierungen (2)

```
#include <setjmp.h>

void find_div_7(int n, jmp_buf *jmpbuf_ptr) {
    if (n % 7 == 0) longjmp(*jmpbuf_ptr, n);
    find_div_7(n + 1, jmpbuf_ptr);
}

int main(void) {
    jmp_buf jmpbuf;          /* type defined in setjmp.h */
    int return_value;

    if ((return_value = setjmp(jmpbuf)) == 0) {
        /* setting up the label for longjmp() lands here */
        find_div_7(1, &jmpbuf);
    }
    else {
        /* returning from a call of longjmp() lands here */
        printf("Answer = %d\n", return_value);
    }
    return 0;
}
```

führt Sprung durch

setzt Sprungmarke

Figure 6.27 Demonstration of the setjmp/longjmp mechanism.

Verwaltung von Lexical Scoping

`lexical ptr`:

verweist auf aktuell gültigen Frame der umgebenden Funktion

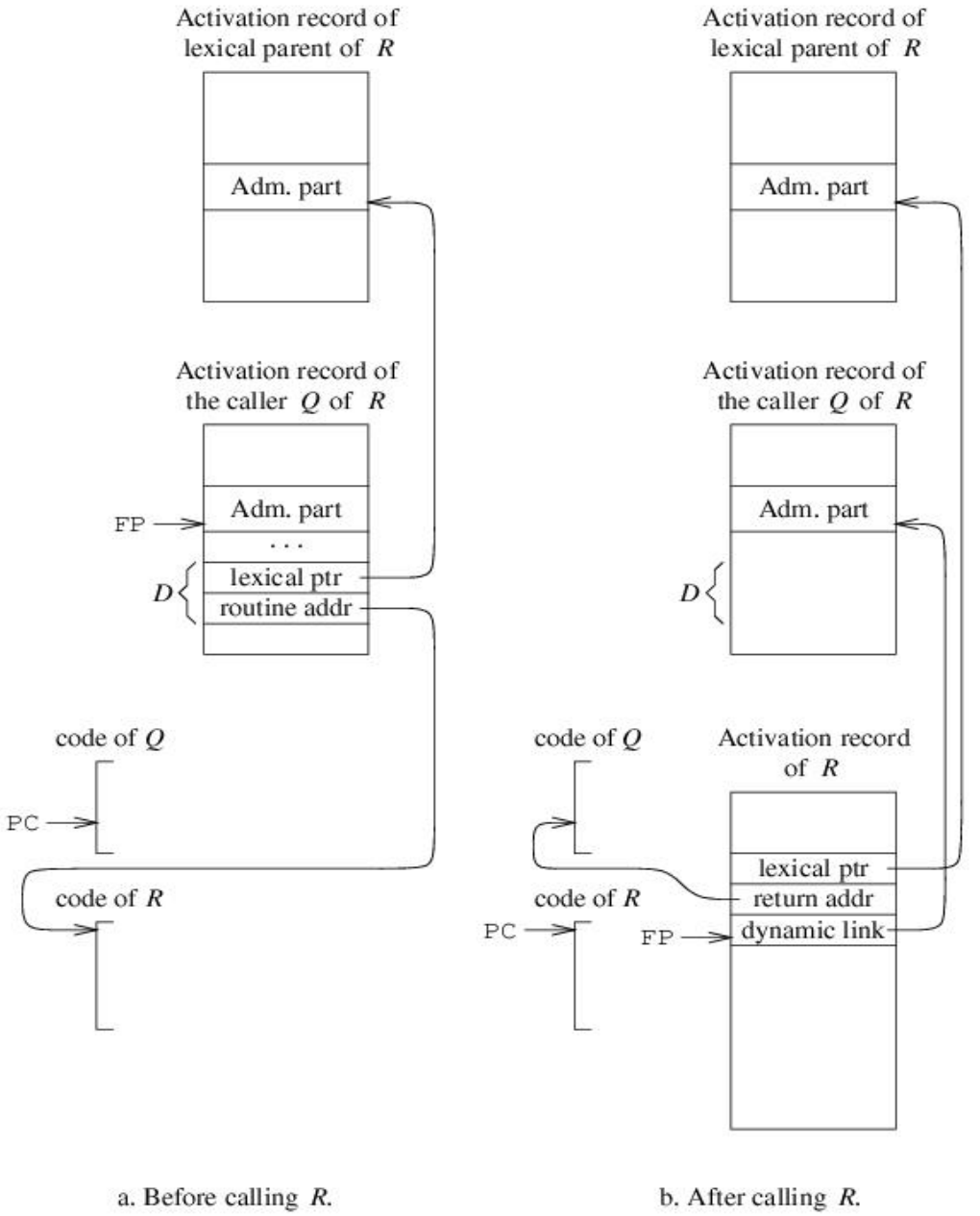


Figure 6.30 Calling a routine defined by the two-pointer routine descriptor *D*.

Zugriffslinks für nichtlokale Variablen (1)

Programm

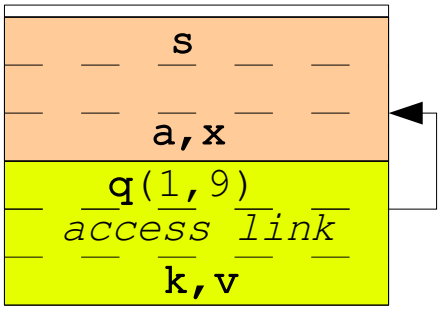
```
void s() {
   int a, x;

   void e(int i, j) {...};

   void q(int m, int n) {
      int k, v;

      void p(int y, int z) {
         int i, j;
      ... }
   ...}
...}
```

Laufzeitstack



Zugriffslinks für nichtlokale Variablen (2)

Programm

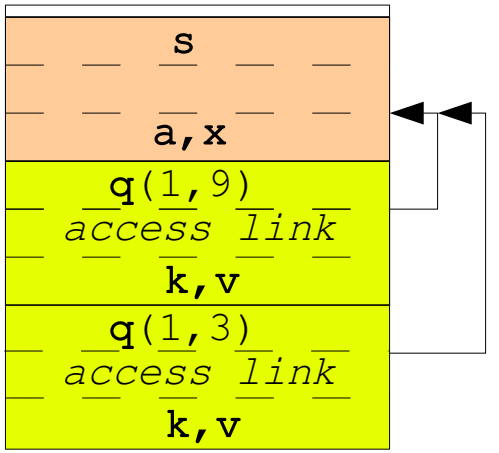
```
void s() {
   int a, x;

   void e(int i, j) {...};

   void q(int m, int n) {
      int k, v;

      void p(int y, int z) {
         int i, j;
      ... }
   ...}
...}
```

Laufzeitstack



Zugriffslinks für nichtlokale Variablen (3)

Programm

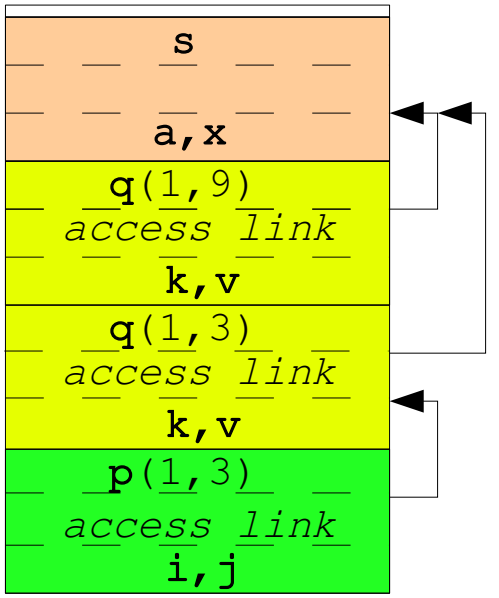
```
void s() {
   int a, x;

   void e(int i, j) {...};

   void q(int m, int n) {
      int k, v;

      void p(int y, int z) {
         int i, j;
         ... }
   ...}
...}
```

Laufzeitstack



Zugriffslinks für nichtlokale Variablen (4)

Programm

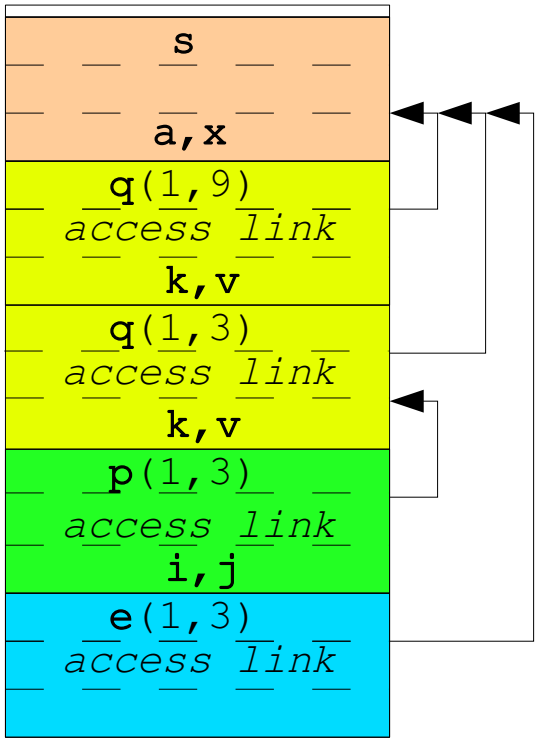
```
void s() {
   int a, x;

   void e(int i, j) {...};

   void q(int m, int n) {
      int k, v;

      void p(int y, int z) {
         int i, j;
         ... }
   ...}
...}
```

Laufzeitstack



Zugriff auf nichtlokale Variablen (1)

```
int i;

void level_0(void) {
    int j;

    void level_1(void) {
        int k;

        void level_2(void) {
            int l;

            ...             /* code has access to i, j, k, l */
            k = l;
            j = l;
        }

        ...                 /* code has access to i, j, k */
        j = k;
    }

    ...                     /* code has access to i, j */
}
```

Figure 6.25 Nested routines in C notation.

Zugriff auf nichtlokale Variablen (2)

```
*(
    *(
        FP
        +
        offset(lexical_pointer)
    )
    +
    offset(k)
) =
*(FP + offset(l))
```

Figure 6.31 Intermediate code for the non-local assignment `k = l`.

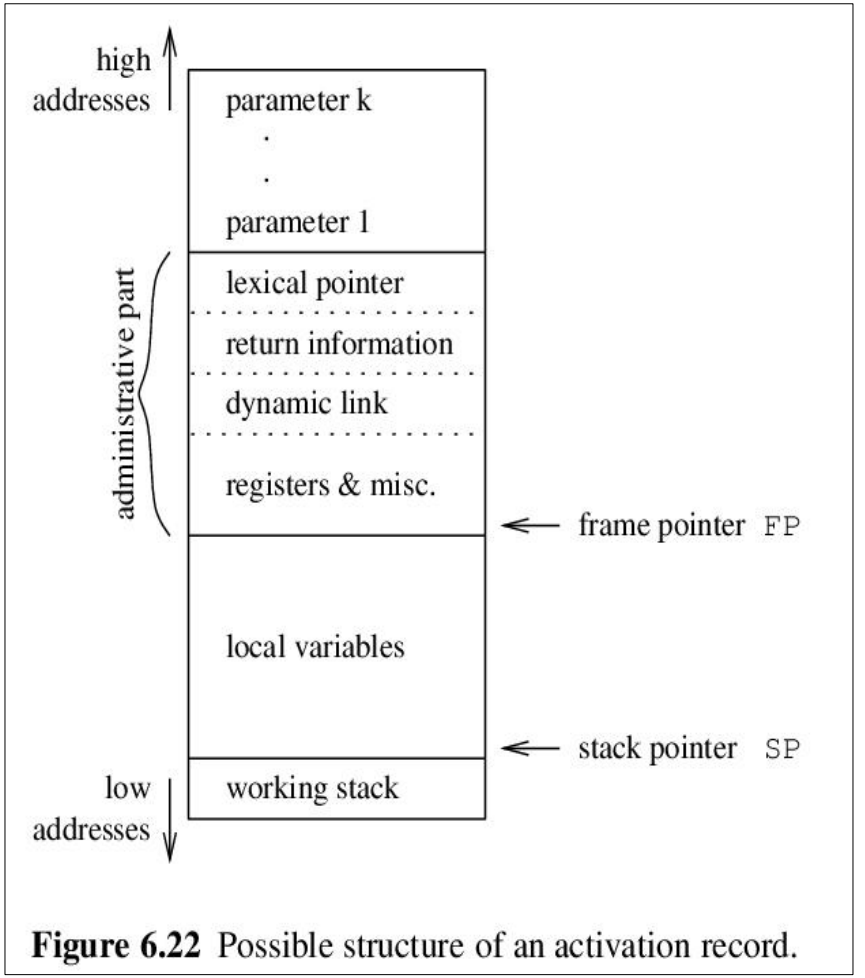


Figure 6.22 Possible structure of an activation record.

Zugriff auf nichtlokale Variablen (3)

Schachtelungsdifferenz=1:

```
*(
    *(
        FP
        +
        offset(lexical_pointer)
    )
    +
    offset(k)
) =
*(FP + offset(l))
```

Figure 6.31 Intermediate code for the non-local assignment `k = l`.

Schachtelungsdifferenz=2:

```
*(
    *(
        *(  FP
            +
            offset(lexical_pointer)
        )
        +
        offset(lexical_pointer)
    )
    +
    offset(j)
) =
*(FP + offset(l))
```

Figure 6.32 Intermediate code for the non-local assignment `j = l`.

Zugriff auf nichtlokale Variablen (4)

```
*(
    *(
        *(   FP
             +
             offset(lexical_pointer)
        )
        +
        offset(lexical_pointer)
    )
    +
    offset(j)
) =
*(FP + offset(l))
```

Figure 6.32 Intermediate code for the non-local assignment `j = l`.

Effizienzproblem:

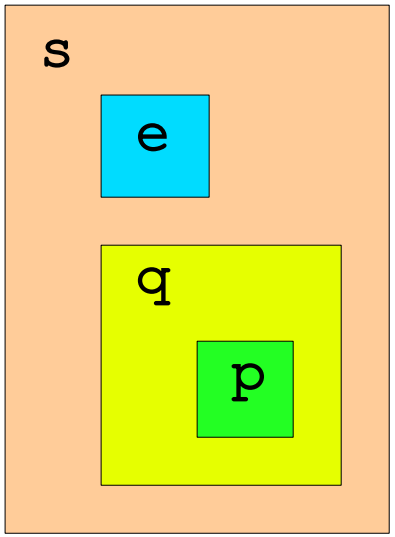
mehrfache indirekte Adressierung beim Zugriff

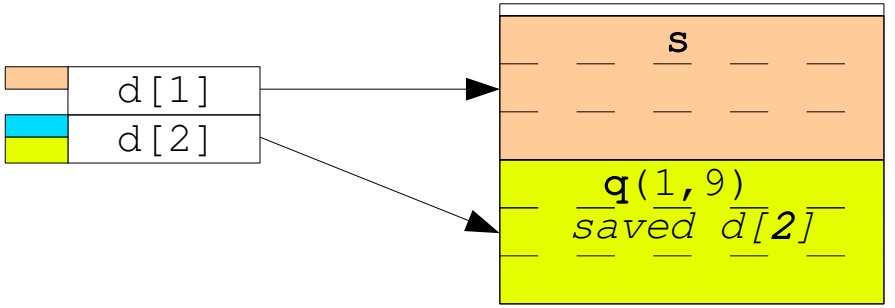
Lösung: “Display” (vgl. Drachenbuch, Abschnitt 7.3.8)

- enthält Tabelle der aktuellen Frames jeder Schachtelungstiefe im Programm
- einfach, wenn Prozeduren keine Parameter sind

Display (1)

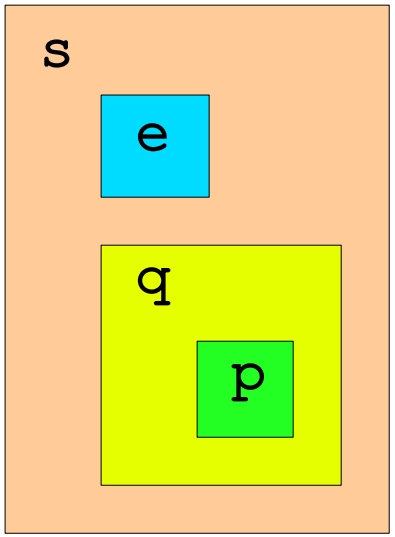
Programmstruktur Display Laufzeitstack

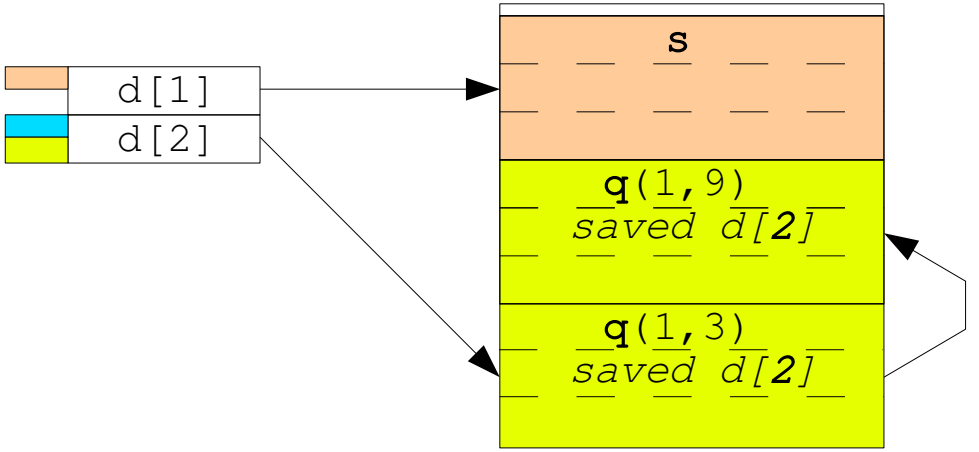




Display (2)

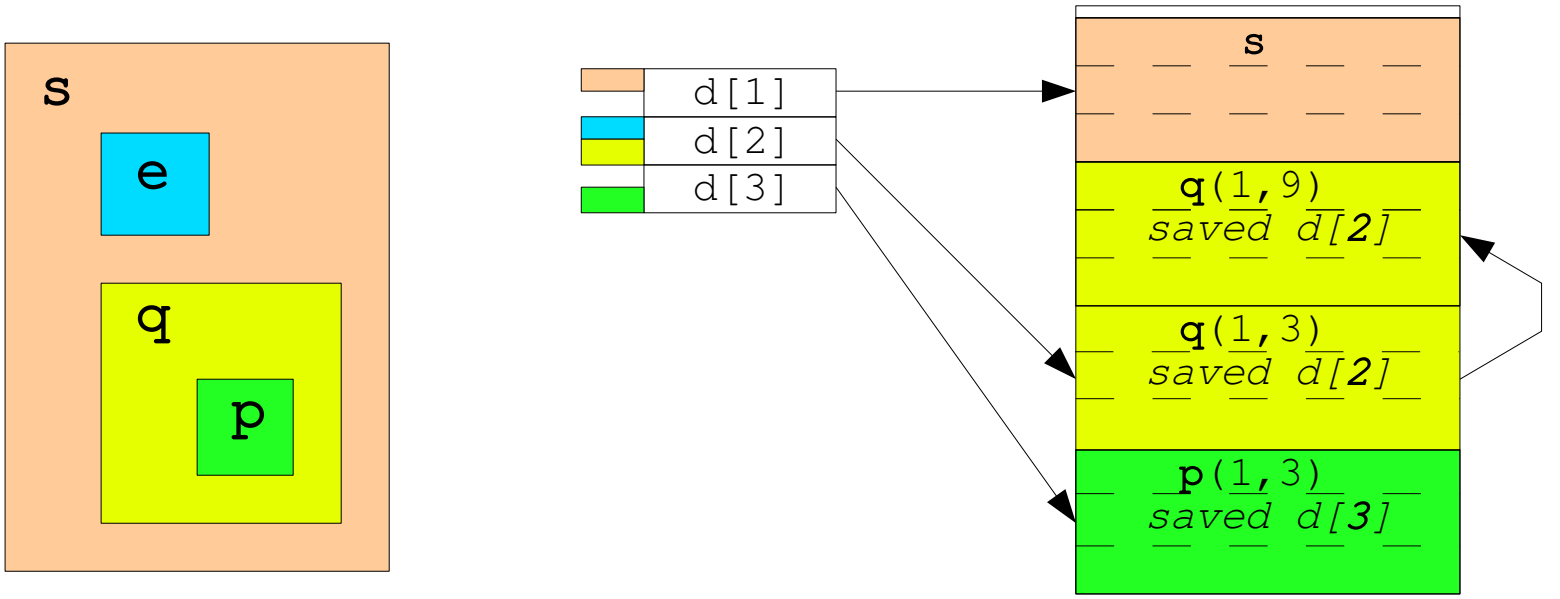
Programmstruktur Display Laufzeitstack





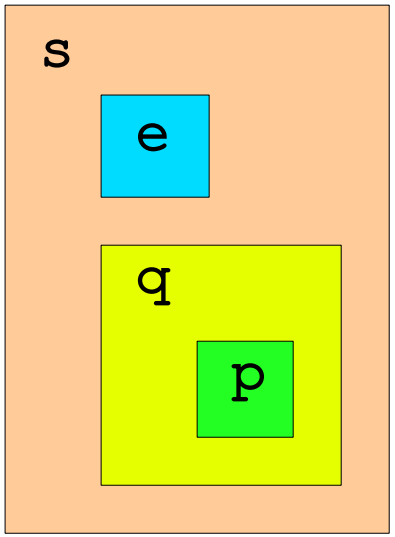
Display (3)

Programmstruktur Display Laufzeitstack

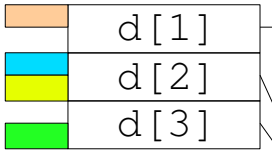


Display (4)

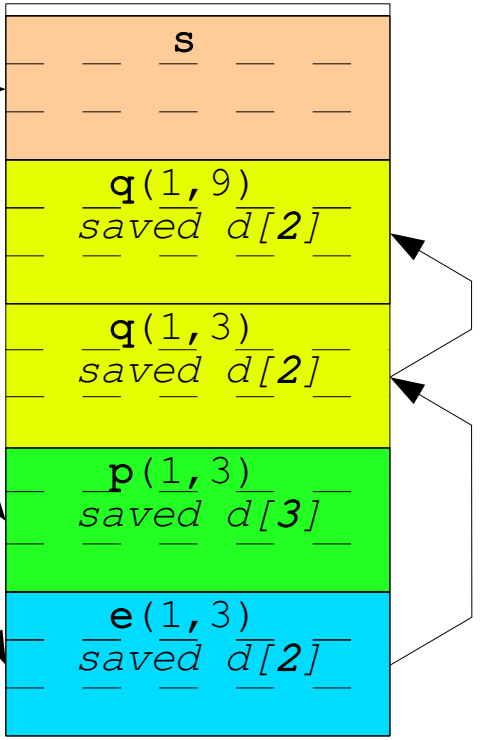
Programmstruktur



Display



Laufzeitstack



Routinen als Argument und Ergebnis

- Standard bei ...
  - funktionalen Sprachen (Haskell, OCaml, LISP)
  - manchen imperativen Sprachen (Pascal: nur als Argument)
- Herausforderung: Kombination von
  - geschachtelten Routinen
  - Zugriff auf statisches Environment (Variablenbindung)
- Routinen als Ergebnis
  - Problem: Environment der zurückgegebenen Routine existiert u.U. nicht mehr auf dem Stack (Pointer-Scope-Regeln verletzt)
  - Lösung: Heapallokation

Übergabe einer geschachtelten Routine (1)

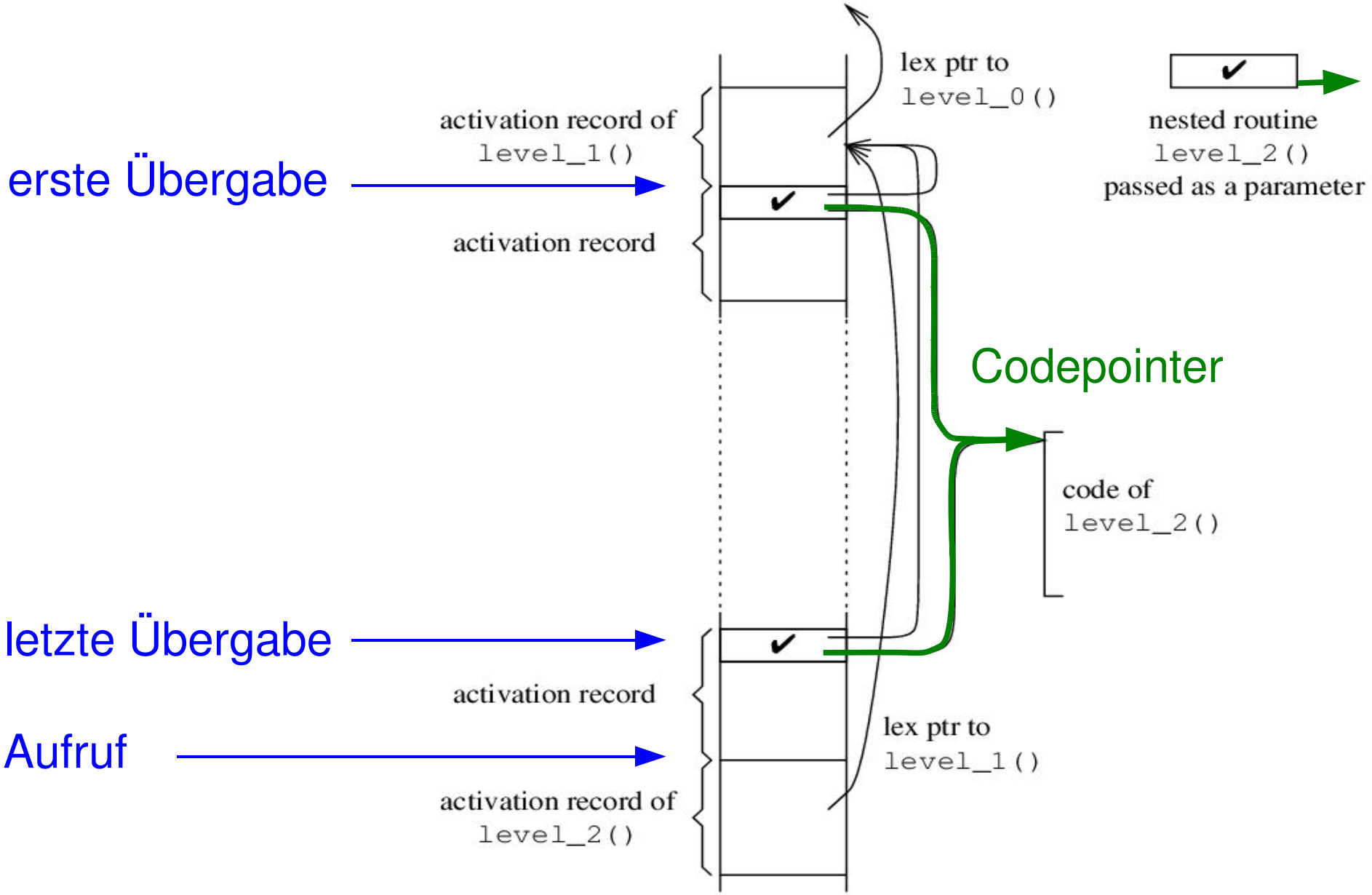


Figure 6.33 Passing a nested routine as a parameter and calling it.

Übergabe einer geschachtelten Routine (2)

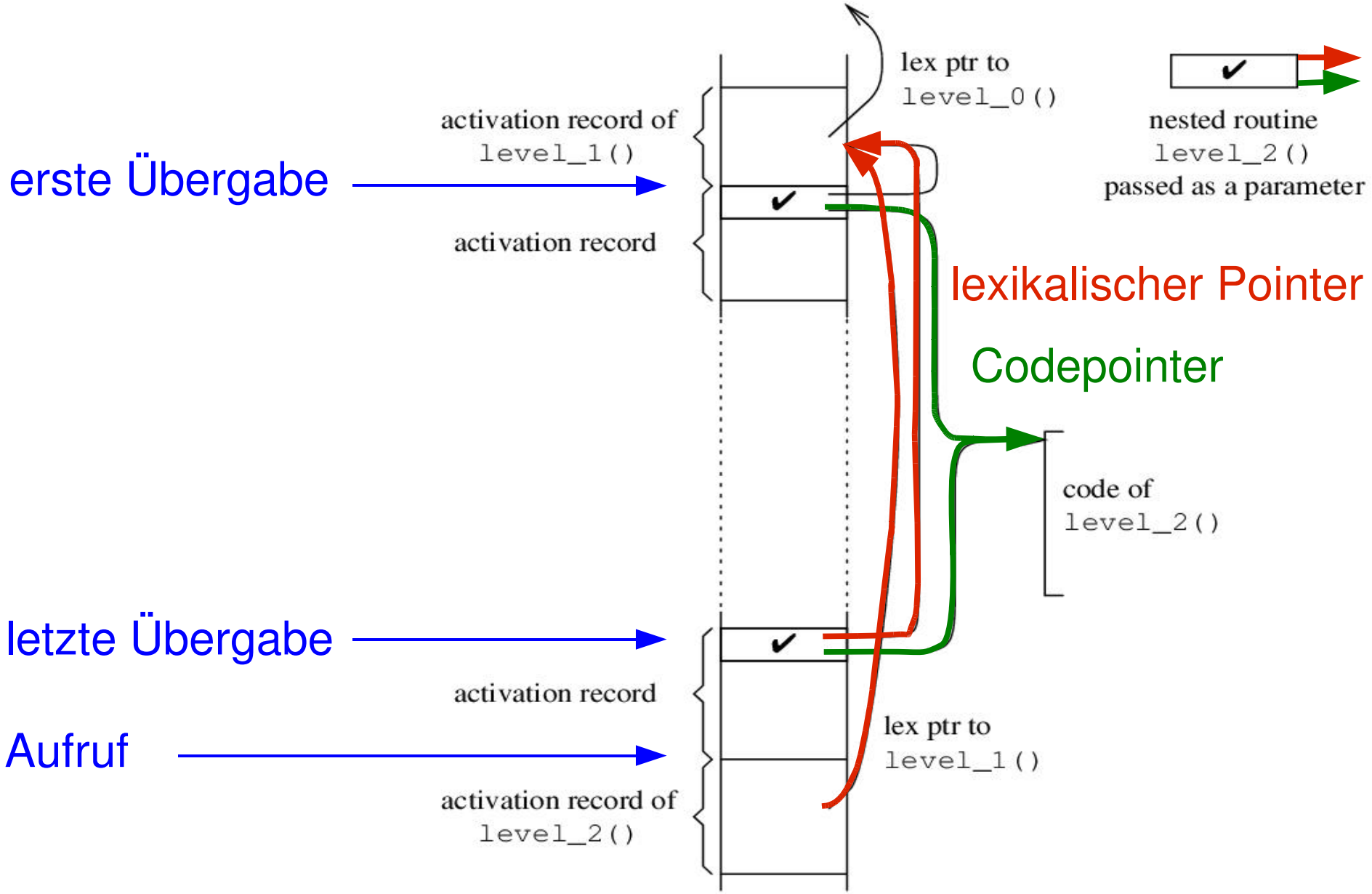


Figure 6.33 Passing a nested routine as a parameter and calling it.

Rückgabe einer geschachtelten Routine

- Pointer-Scope-Regel:
  - (zeitlicher) Existenzbereich eines Pointers muss in seinem (zeitlichen) Scope liegen
- Routine als Argument:
  - Stackallokation möglich, weil Environment der übergebenen Routine auf dem Stack existiert
  - Existenzbereich des lexikalischen Zeigers liegt innerhalb seines Scopes
- Routine als Ergebnis:
  - Pointer-Scope-Regel verletzt, wenn Aktivierungssegmente auf dem Stack lägen
  - Lösung: Heapallokation (Zeigerscope unendlich)

Sprung aus einer geschachtelten Routine (1)

```
void level_0(void) {
    void level_1(void) {
        void level_2(void) {
            ...
            goto L_1;
            ...
        }
        ...
    L_1:...
        ...
    }
    ...
}
```

Figure 6.26 Example of a non-local goto.

Deskriptor mit Sprunginformation

- Framepointer der letzten Aktivierung von `level_1`
- Codepointer zu `L_1`

Sprung aus einer geschachtelten Routine (2)

Label-Deskriptor: Speicherung und Zugriff

```
void level_1(void) {
    non_local_label_descriptor L_1_as_a_value = {
        FP_of_this_activation_record(),  /* FP of level_1() */
        L_1                              /* code address of L_1 */
    };

    void level_2(void) {

        ...
        non_local_goto(L_1_as_a_value);      /* goto L_1; */
        ...
    }

    ...
L_1:...
    ...
}
```

Figure 6.34 Possible code for the construction of a label descriptor.

Partielle Parametrisierung (1)

- Routinen können vor der Übergabe/Rückgabe spezialisiert werden
- Werte bestimmter Argumente werden vor dem eigentlichen Aufruf der Routine festgelegt
- Bsp. (in Haskell): partielle Parametrisierung einer Additionsfunktion übergeben als Argument bzw. Ergebnis

```
add (x,y) = x+y

example1 x = let twice (f,z) = f (f z)
             in twice (\y -> add (1,y), x)

example2 x = let f d = \y -> add (d,y)
                 g   = f 1
             in g x
```

Partielle Parametrisierung (2)

im imperativen Stil (Syntax C-ähnlich)

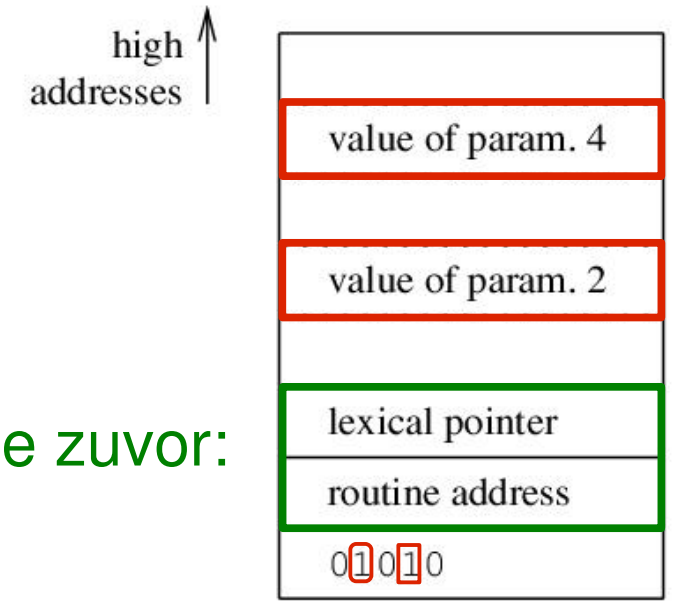
```
extern int add(int i, int j);

int (*inc) (int i);

int main(void) {
    ...
    inc = add(, 1);  // zweiter Parameter festgelegt
    ...
    printf("%d\n", inc(5));
    ...
}
```

Partielle Parametrisierung (3)

erweiterter Deskriptor



Werte instanzierter Parameter (andere Positionen undefiniert)

wie zuvor:

neu: instanzierte Positionen

Figure 6.35 A closure for a partially parameterized routine.

Partielle Parametrisierung (4)

nachdem alle Parameter instanziert sind:

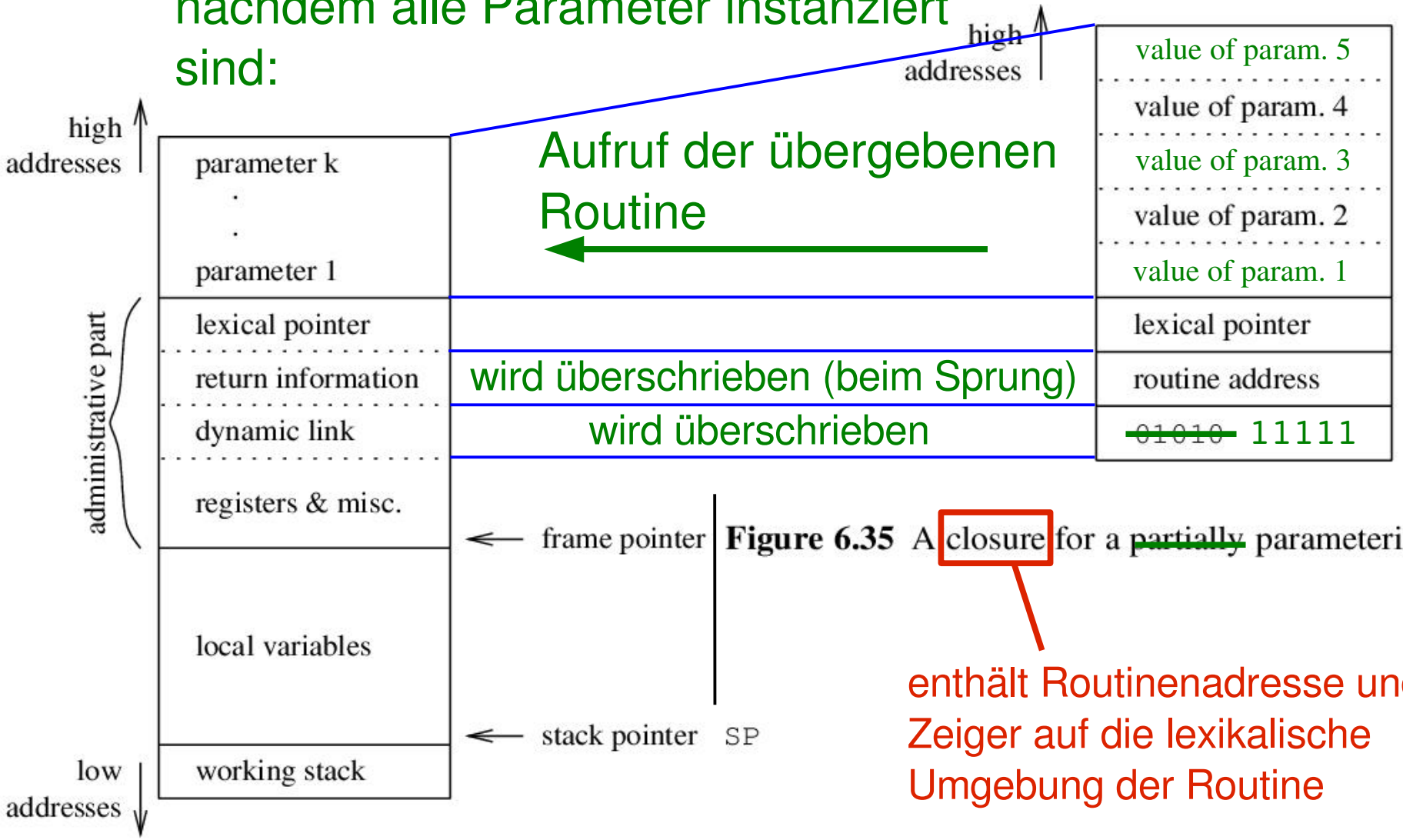


Figure 6.35 A closure for a ~~partially~~ parameterized routine.

Figure 6.22 Possible structure of an activation record.

Lambda-Lifting (1)

geschachteltes Programm

```
int i;

void level_0(void) {
    int j;

    void level_1(void) {
        int k;

        void level_2(void) {
            int l;

            ...
            k = l;
            j = l;
        }

        ...
        j = k;
    }

    ...
}
```

→

flaches Programm

```
int i;

void level_2(int *j, int *k) {
    int l;

    ...
    *k = l;
    *j = l;
}

void level_1(int *j) {
    int k;

    ...
    level_2(j, &k);
    A(level_2);

}

void level_0(void) {
    int j;

    ...
    level_1(&j);
}
```

Lambda-Lifting (2)

```
int i;

void level_2(int *j, int *k) {
    int l;

    ...                                 /* code has access to i, *j, *k, l */
    *k = l;
    *j = l;
}

void level_1(int *j) {
    int k;

    ...                                 /* code has access to i, *j, k */
    level_2(j, &k);                     /* was: level_2(); */
    A(level_2);                         /* level_2() as a parameter: */
                                        /* this is a problem */
}

void level_0(void) {
    int j;

    ...                                 /* code has access to i, j */
    level_1(&j);                        /* was: level_1(); */
}
```

Figure 6.36 Lambda-lifted routines in C notation.

Heap-Allokation (1)

```
int i;

void level_2(int *j, int *k) {
    int l;

    ...
    *k = l;
    *j = l;
}

void level_1(int *j) {
    int k;

    ...
    level_2(j, &k);
    A(level_2);

}

void level_0(void) {
    int j;

    ...
    level_1(&j);
}
```

```
int i;

void level_2(int *j, int *k) {
    int l;

    ...
    *k = l;
    *j = l;
}

void level_1(int *j) {
    int *k = (int *)malloc(sizeof (int));

    ...
    level_2(j, k);
    A(closure(level_2, j, k));
}

void level_0(void) {
    int *j = (int *)malloc(sizeof (int));

    ...
    level_1(j);
}
```

Heap-Allokation (2)

```
int i;

void level_2(int *j, int *k) {
    int l;

    ...                                 /* code has access to i, *j, *k, l */
    *k = l;
    *j = l;
}

void level_1(int *j) {
    int *k = (int *)malloc(sizeof (int));

    ...                                 /* code has access to i, *j, *k */
    level_2(j, k);                      /* was: level_2(); */
    A(closure(level_2, j, k));          /* was: A(level_2); */
}

void level_0(void) {
    int *j = (int *)malloc(sizeof (int));

    ...                                 /* code has access to i, *j */
    level_1(j);                         /* was: level_1(); */
}
```

Figure 6.37 Lambda-lifted routines with additional heap allocation in C notation.

Boolesche Ausdrücke für Kontrollfluss

- Boolesche Ausdrücke oft nur zur Steuerung des Kontrollflusses
- Optimierungen
  - vermeide Konversionen

    (1) Bedingungsregister (nach Vergleich) → Boolescher Wert

    (2) Boolescher Wert → Bedingungsregister (für Sprung)
  - Kurzschlussauswertung (“lazy”-Auswertung)
    - keine Auswertung von `a` bei `(False && a)`
    - keine Auswertung von `a` bei `(True || a)`
  - Vertauschung der Sprungziele statt Auswertung von `!` (not)

Codegenerierung für Kontrollflussanweisungen

- jeder Ausdruck hat zwei Sprungziele, "`False_label`" und "`True_label`", enthalten entweder

  (a) Sprungadresse

  oder

  (b) `No_label` (Code dafür folgt direkt, kein Sprung nötig)

- Codegenerierung top-down im Ausdrucksbaum

- an jedem Knoten
  - einfügen temporärer Sprungziele
  - Codegenerierung für Teilausdrücke mit individuell modifizierten Sprungzielen

Atomare Bedingungen (Vergleiche)

```
IF True label /= No label:
    Emit ("IF condition register THEN GOTO" True label);
    IF False label /= No label:
        Emit ("GOTO" False label);
ELSE True label = No label:
    IF False label /= No label:
        Emit ("IF NOT condition register THEN GOTO"
            False label);
```

Kontrollfluss für logisches “Und”

“Kurzschlussauswertung” von `(Bed1 && Bed2)`

- falls `Bed1` falsch ist, wird sofort zum Sprungziel “False_label” gesprungen
- falls `Bed1` wahr ist, erfolgt kein Sprung (“`No_label`”), sondern der Code für `Bed2` wird ausgeführt

```
Generate code for Boolean control expression
    (Node .left, No label, False label);
Generate code for Boolean control expression
    (Node .right, True label, False label);
```

Kontrollflussausdrücke, Knotenoperation

```
PROCEDURE Generate code for Boolean control expression (
    Node, True label, False label
):
    SELECT Node .type:
        CASE Comparison type:                          // <, >, ==, etc. in C
            Generate code for comparison expression (Node .expr);
            // The comparison result is now in the condition register
            IF True label /= No label:
                Emit ("IF condition register THEN GOTO" True label);
                IF False label /= No label:
                    Emit ("GOTO" False label);
            ELSE True label = No label:
                IF False label /= No label:
                    Emit ("IF NOT condition register THEN GOTO"
                        False label);
        CASE Lazy and type:                            // the && in C
            Generate code for Boolean control expression
                (Node .left, No label, False label);
            Generate code for Boolean control expression
                (Node .right, True label, False label);
        CASE ...
        CASE Negation type:                            // the ! in C
            Generate code for Boolean control expression
                (Node, False label, True label);
```

Fehler im Buch, richtig: `Node .left`

Figure 6.38 Code generation for Boolean expressions.