program specialization. If the goal were maximal efficiency it would be natural to let `mix` generate programs in a lower-level language [120].

4.3 Partial evaluation and compilation

In this section we first show by a concrete example that a partial evaluator can be used to compile, given an interpreter and a source program. We then show that this remarkable fact is a simple consequence of the mix equation as presented above and from the definitions of interpreters and compilers from Section 3.1. This result is known as *the first Futamura projection*.

4.3.1 An interpreter for Turing machine programs

This section presents a concrete program that will be used to illustrate several points later in the chapter. The program is an interpreter for a Turing machine (Post's variant) with tape alphabet `A = {0,1,B}`, where `B` stands for 'blank'. A Turing program `Q` is a list $(I_0\ I_1\ \ldots I_n)$ of instructions each of form

`right`, `left`, `write a`, `goto i`, or `if a goto i`

A computational state consists of a current instruction I_i about to be executed, and an infinite tape of squares a_i:

$\ldots a_{-2}\ a_{-1}\ a_0\ a_1\ a_2\ \ldots$

Only finitely many of the squares contain symbols a_i not equal to `B`; a_0 is called the *scanned square*. Instruction effects: `write a` changes a_0 to `a`, `right` and `left` change the scanning point, and `if a goto i` causes the next control point to be I_i in case a_0 `= a`; in all other cases the next control point is the following instruction (if any).

An example program `Q` in the Turing language is given in Figure 4.3. The input to this program is $a_0a_1\ \ldots a_n \in \{0,1\}*$, and the initial tape contains `B` in all other positions, that is, $a_{n+1}, a_{n+2}, \ldots$, and $a_{-1}, a_{-2}, \ldots$. Program output is the final value of $a_0\ a_1\ \ldots$ (at least up to and including the last non–blank symbol) and is produced when there is no next instruction to be executed. Note that a different square may be scanned on termination than at the beginning.

```
0: if 0 goto 3
1: right
2: goto 0
3: write  1
```

Figure 4.3: A Turing machine program.

The program finds the first `0` to the right on the initial tape and converts it to `1` (and goes into an infinite loop if none is found). If the input to `Q` is `110101`, the output will be `1101`.

The Turing interpreter in Figure 4.4 has a variable `Q` for the whole Turing program, and the control point is represented via a suffix `Qtail` of `Q` (the list of instructions remaining to be executed). The tape is represented by variables `Left`, `Right` with values in `A*`, where `Right` equals $a_0\ a_1\ a_2\ \ldots$ (up to and including the last non–blank symbol) and `Left` similarly represents $a_{-1}\ a_{-2}\ a_{-3}\ \ldots$. Note that the order is reversed.

```
read (Q, Right);
init:     Qtail := Q;  Left := '();

loop:     if Qtail = '() goto stop else cont;
cont:     Instruction  :=  first_instruction(Qtail);
          Qtail        :=  rest(Qtail);
          Operator     :=  hd(tl(Instruction));

          if Operator = 'right goto do-right else cont1;
cont1:    if Operator = 'left  goto do-left  else cont2;
cont2:    if Operator = 'write goto do-write else cont3;
cont3:    if Operator = 'goto  goto do-goto  else cont4;
cont4:    if Operator = 'if    goto do-if    else error;

do-right: Left      := cons(firstsym(Right), Left);
          Right     := tl(Right); goto loop;
do-left:  Right     := cons(firstsym(Left), Right);
          Left      := tl(Left); goto loop;
do-write: Symbol    := hd(tl(tl(Instruction)));
          Right     := cons(Symbol,tl(Right)); goto loop;
do-goto:  Nextlabel := hd(tl(tl(Instruction)));
          Qtail     := new_tail(Nextlabel, Q); goto loop;
do-if:    Symbol    := hd(tl(tl(Instruction)));
          Nextlabel := hd(tl(tl(tl(tl(Instruction)))));
          if Symbol = firstsym(Right) goto jump else loop;

jump:     Qtail     := new_tail(Nextlabel,Q); goto loop;

error:    return ('syntax-error: Instruction);

stop:     return right;
```

Figure 4.4: Turing machine interpreter written in L.

The interpreter uses some special base functions. These are `new_tail`, which takes a label `lab` and the program `Q` as arguments and returns the part (suffix) of the

program beginning with label `lab`; `first_instruction`, which returns the first instruction from an instruction sequence; and `rest`, which returns all but the first instruction from an instruction sequence. Moreover, we need a special version `firstsym` of `hd` for which `firstsym () = B`, and we assume that `tl ()` is `()`.

Example 4.2 Let `Q` be `(0: if 0 goto 3  1: right  2: goto 0  3: write 1)`. Then

```
⟦int⟧L [Q, 110101]      =  1101
new_tail(2, Q)          =  (2: goto 0  3: write 1)
first_instruction(Q)    =  (0: if 0 goto 3)
rest(Q)                 =  (1: right  2: goto 0  3: write 1)
```

are some typical values of these auxiliary functions. □

Time analysis. The Turing interpreter in Figure 4.4 executes between 15 and 28 operations per executed command of `Q`, where we count one operation for each assignment, `goto` or base function call.

4.3.2 The Futamura projections

Futamura was the first researcher to realize that self-application of a partial evaluator can in principle achieve compiler generation [92]. Therefore the equations describing compilation, compiler generation, and compiler generation are now called the *Futamura projections*.

```
target    =  ⟦mix⟧L [int, source program]
compiler  =  ⟦mix⟧L [mix, int]
cogen     =  ⟦mix⟧L [mix, mix]
```

Although easy to verify, it must be admitted that the intuitive significance of these equations is hard to see. In the remainder of this chapter we shall give some example target programs, and a compiler derived from the interpreter just given.

4.3.3 Compilation by the first Futamura projection

In this section we shall show how we can compile programs using only an interpreter and the program specializer. We start by verifying the *first Futamura projection*, which states that specializing an interpreter with respect to a source program has the effect of compiling the source program. Let `int` be an `S`-interpreter written in `L`, let `s` be an `S`-program, and `d` its input data. The equation is proved by:


```
⟦s⟧S d  =  ⟦int⟧L [s,d]                 by the definition of
                                            an interpreter
        =  ⟦(⟦mix⟧L [int,s] )⟧L d       by the mix equation
        =  ⟦target⟧L d                  by naming the residual
                                            program: target
```

These equations state nothing about the quality of the target program, but in practice it can be quite good. Figure 4.5 shows a target program generated from the above interpreter (Figure 4.4) and the source program `s = (0: if 0 goto 3  1: right  2: goto 0  3: write 1)`. Here we just present the result; a later section will show how it was obtained.

```
read (Right);
lab0: Left := '();
      if '0 = firstsym(Right) goto lab2 else lab1;
lab1: Left := cons(firstsym(Right), Left);
      Right := tl(Right);
      if '0 = firstsym(Right) goto lab2 else lab1;
lab2: Right := cons('1, tl(Right));
      return(Right);
```

Figure 4.5: A `mix`*-generated target program.*

Notice that the target program is written in the same language as the interpreter; this comes immediately from the `mix` equation. On the other hand, this target program's *structure* more closely resembles that of the source program from which it was derived than that of the interpreter. Further, it is composed from bits and pieces of the interpreter, for example `Left := cons(firstsym(Right), Left)`. Some of these are *specialized* with respect to data from the source program, e.g. `if '0 = firstsym(Right) goto lab2 else lab1`. This is characteristic of `mix`-produced target programs.

Time analysis. We see that the target program (Figure 4.5) has a quite natural structure. The main loop in the target program takes 8 operations while the interpreter takes 61 operations to interpret the main loop of the source program, so the target program is nearly 8 times faster than the interpreter when run on this source program.

4.4 Program specialization techniques

We now describe basic principles sufficient for program specialization; a concrete algorithm will be given in a later section.

$$((pp, v_s), v_d) \Rightarrow ((pp', v'_s), v'_d)$$

This is also a transition, but one with *specialized control points* (pp, v_s) and (pp', v'_s), each incorporating some static data. The runtime data are v_d, v'_d, the result of the dynamic projections.

Fundamental concepts revisited
Residual code generation amounts to finding commands or a function or procedure call which syntactically specifies the transition from v_d to v'_d. If $v_d = v'_d$ then *transition compression* may be possible since no residual code beyond at most a control transfer need be generated. (For flow charts, this happens if the basic block begun by pp contains no dynamic expressions or commands.) Finally, we have seen the congruence condition to be needed for code generation. In the current context this becomes: v'_s must be functionally determined by v_s in every transition.

4.12 Exercises

Exercise 4.1 Write a program and choose a division such that partial evaluation without transition compression terminates, and partial evaluation with transition compression on the fly (as described in this chapter) loops. □

Exercise 4.2 The purpose of this exercise is to investigate how much certain extensions to the flow chart language would complicate partial evaluation. For each construction, analyse possible problems and show the specialization time computations and the code generation

1. for loop,
2. while loop,
3. case/switch conditional,
4. computed `goto`, `cgoto` ⟨Expr⟩, where Expr evaluates to a natural number = a label,
5. `gosub ... return`.

Do any of the above constructions complicate the binding-time analysis? □

Exercise 4.3 At the end of Section 4.2.2 it is mentioned that mix could generate residual programs in a low-level language, e.g. machine code.

1. Write the mix equation for a mix that generates machine code.
2. Do the Futamura projections still hold?
3. What are the consequences for compiler generation?

□

Exercise 4.4 Consider the mix-generated program in Figure 4.5. The program is suboptimal in two ways; discuss how to revise the partial evaluation strategy to obtain an optimal residual program in this particular example.

1. The same conditional statement appears twice.
2. The assignments to the variable `Left` do not contribute to the final answer.

Would the proposed revisions have any adverse effects? □

Exercise 4.5 Specialize the Turing interpreter with respect to to following program:

```
0: if B goto 3
1: right
2: goto 0
3: write 1
4: if B goto 7
5: left
6: goto 4
7: write 1
```

□

Exercise 4.6 The mix equation and the Futamura projections as presented in this chapter gloss over the fact that partial evaluation (here) consists of a binding-time analysis phase and a specialization phase. Refine the mix-equation and the Futamura projections to reflect this two-phase approach. □

Exercise 4.7 Will specialization of the Turing interpreter (Figure 4.4) with respect a program `p` terminate for all Turing programs `p`? □

Exercise 4.8 Use the algorithm in Section 4.4.6 to determine a congruent division for the Turing interpreter when the division for the input variables (`Q, Right`) is

1. (S, D),
2. (D, S).

□

Exercise 4.9 Write a binding time analysis algorithm that computes a pointwise division. □

Exercise 4.10 Write a binding time analysis algorithm that computes a polyvariant division. □

Exercise 4.11 The binding time analysis algorithm from Section 4.4.6 is not likely to be very efficient in practice. Construct an efficient algorithm to do the same job. □