[next] [prev] [prev-tail] [tail] [up]

1.3 Programs

Choice of a Programming Language
Deterministic Programs
Computations
Nondeterministic Programs
Guessing in Programs
Configurations of Programs

Our deep dependency on the processing of information brought about the deployment of programs in an ever increasing array of applications. Programs can be found at home, at work, and in businesses, libraries, hospitals, and schools. They are used for learning, playing games, typesetting, directing telephone calls, providing medical diagnostics, forecasting weather, flying airplanes, and for many other purposes.

To facilitate the task of writing programs for the multitude of different applications, numerous programming languages have been developed. The diversity of programming languages reflects the different interpretations that can be given to information. However, from the perspective of their power to express computations, there is very little difference among them. Consequently, different programming languages can be used in the study of programs.

The study of programs can benefit, however, from fixing the programming language in use. This enables a unified discussion about programs. The choice, however, must be for a language that is general enough to be relevant to all programs but primitive enough to simplify the discussion.

Choice of a Programming Language

Here, a *program* is defined as a finite sequence of instructions over some domain D. The domain D, called the *domain of the variables*, is assumed to be a set of elements with a distinguished element, called the *initial value of the variables*. Each of the elements in D is assumed to be a possible assignment of a value to the variables of the program. The sequence of instructions is assumed to consist of instructions of the following form.

a. Read instructions of the form

```
read x
```

where x is a variable.

b. Write instructions of the form

```
write x
```

where x is a variable.

c. Deterministic assignment instructions of the form

```
y := f(x_1, . . . , x_m)
```

where $x_1, \ldots, x_m$, and y are variables, and f is a function from D^m to D.

d. Conditional if instructions of the form

```
if Q(x1, . . . , xm) then I
```

where I is an instruction, $x_1, \ldots, x_m$ are variables, and Q is a predicate from D^m to {false, true}.

e. Deterministic looping instructions of the form

```
do
    α
until Q(x1, . . . , xm)
```

where α is a nonempty sequence of instructions, $x_1, \ldots, x_m$ are variables, and Q is a predicate from D^m to {false, true}.

f. Conditional accept instructions of the form

```
if eof then accept
```

g. Reject instructions of the form

```
reject
```

h. Nondeterministic assignment instructions of the form

```
x := ?
```

where x is a variable.

i. Nondeterministic looping instructions of the form

```
do
    α1
or
    α2
or
    ⋮
or
    αk
until Q(x1, . . . , xm)
```

where $k \geq 2$, each of $\alpha_1, \ldots, \alpha_k$ is a nonempty sequence of instructions, $x_1, \ldots, x_m$ are variables, and Q is a predicate from D^m to {false, true}.

In each program the domain D of the variables is assumed to have a representation over some alphabet. For instance, D can be the set of natural numbers, the set of integers, and any finite set of elements. The functions f and predicates Q are assumed to be from a given "built-in" set of computable functions and predicates (see Section 1.4 and Church's thesis in Section 4.1).

In what follows, the domains of the variables will not be explicitly noted when their nature is of little significance. In addition, expressions in infix notations will be used for specifying functions and predicates.

Programs without nondeterministic instructions are called *deterministic* programs, and programs with nondeterministic instructions are called *nondeterministic* programs.

Example 1.3.1 The program P_1 in Figure 1.3.1(a)

```
read x
y := 0
z := 1
do
    y := y + 1
   z := z + 1
until z = x
read y
if eof then accept
reject
```

(a)

```
do
    read x
or
    y := ?
   write y
until y = x
if eof then accept
```

(b)

Figure 1.3.1 (a) A deterministic program. (b) A nondeterministic program.

is an example of a deterministic program, and the program P_2 in Figure 1.3.1(b) is an example of a nondeterministic program. The set of natural numbers is assumed for the domains of the variables, with 0 as initial value.

The program P_1 uses three variables, namely, x, y, and z. There are two functions in this program. The constant function $f_1() = 0$, and the unary function $f_2(n) = n + 1$ of addition by one. The looping instruction uses the binary predicate of equality.

The program P_2 uses two nondeterministic instructions. One of the nondeterministic instructions is an assignment instruction of the form "y := ?"; the other is a looping instruction of the form "**do . . . or . . . until . . .** " □

An *input* of a given program is a sequence of elements from the domain of the variables of the program. Each element in an input of a program is called an *input value*.

Example 1.3.2 The programs of Example 1.3.1 (see Figure 1.3.1) can have any input that is a finite sequence of natural numbers. An input of the form "1, 2, 3, 4" consists of four input values, and an input of the form " " contains no input value.

The sequence "1, 2, 3, . . . " cannot be an input for the programs because it is not a finite sequence. □

An *execution sequence* of a given program is an execution on a given input of the instructions according to their semantics. The instructions are executed consecutively, starting with the first instruction. The variables initially hold the initial value of the variables.

Deterministic Programs

Deterministic programs have the property that no matter how many times they are executed on a given input, the executions are always in exactly the same manner. Each instruction of a deterministic program fully specifies the operations to be performed. In contrast, nondeterministic instructions provide only partial specifications for the actions.

An execution of a read instruction **read** x reads the next input value to x. An execution of a write instruction **write** x writes the value of x.

The deterministic assignment instructions and the conditional if instructions have the conventional semantics.

An execution of a deterministic looping instruction **do** α **until** $Q(x_1, \ldots, x_m)$ consists of repeatedly executing α and checking the value of $Q(x_1, . . . , x_m)$. The execution of the looping instruction is terminated upon detecting that the predicate $Q(x_1, . . . , x_m)$ has the value true. If $Q(x_1, . . . , x_m)$ is the constant true, then only one iteration is executed. On the other hand, if $Q(x_1, . . . , x_m)$ is the constant false, then the looping goes on forever, unless the execution terminates in α.

A conditional accept instruction causes an execution sequence to halt if executed after all the input is consumed, that is, after reaching the end of input file (*eof* for short). Otherwise the execution of the instruction causes the execution sequence to continue at the code following the instruction. Similarly, an execution sequence also halts upon executing a reject instruction, trying to read beyond the end of the input, trying to transfer the control beyond the end of the program, or trying to compute a value not in the domain of the variables (e.g., trying to divide by 0).

Example 1.3.3 Consider the two programs in Figure 1.3.2.

```
do                          do
    if eof then accept          read value
    read value                  write value
    write value             until value < 0
until false                 if eof then accept
          (a)                         (b)
```

Figure 1.3.2 Two deterministic programs.

Assume that the programs have the set of integers for the domains of their variables, with 0 as initial value.

For each input the program in Figure 1.3.2(a) has one execution sequence. In each execution sequence the program provides an output that is equal to the input. All the execution sequences of the program terminate due to the execution of the conditional accept instruction.

On input "1, 2" the execution sequence repeatedly executes for three times the body of the deterministic looping instruction. During the first iteration, the execution sequence determines that the predicate *eof* has the value false. Consequently, the execution sequence ignores the accept command and continues by reading the value 1 and writing it out. During the second iteration the execution sequence verifies again that the end of the input has not been reached yet, and then the execution sequence reads the input value 2 and writes it out. During the third iteration, the execution sequence terminates due to the accept command, after determining a true value for the predicate *eof* .

The execution sequences of the program in Figure 1.3.2(b) halt due to the conditional accept instruction, only on inputs that end with a negative value and have no negative values elsewhere (e.g., the input "1, 2, -3"). On inputs that contain no negative values at all, the execution sequences of the program halt due to trying to read beyond the end of the input (e.g., on input "1, 2, 3"). On inputs that have negative values before their end, the execution sequences of the program halt due to the transfer of control beyond the end of the program (e.g., on input "-1, 2, -3"). □

Intuitively, an **accept** can be viewed as a halt command that signals a successful completion of a program execution, where the **accept** can be executed only after the end of the input is reached. Similarly, a **reject** can be viewed as a halt instruction that signals an unsuccessful completion of a program execution.

The requirement that the accept commands be executed only after reading all the input values should cause no problem, because each program can be modified to satisfy this condition. Moreover, such a constraint seems to be natural, because it forces each program to check all its input values before signaling a success by an accept command. Similarly, the requirement that an execution sequence must halt upon trying to read beyond the end of an input seems to be natural. It should not matter whether the reading is due to a read instruction or to checking for the *eof* predicate.

It should be noted that the predicates $Q(x_1, \ldots, x_m)$ in the conditional if instructions and in the looping instructions cannot be of the form *eof* . The predicates are defined just in terms of the values of the variables $x_1, \ldots, x_m$, not in terms of the input.

Computations

Programs use finite sequences of instructions for describing sets of infinite numbers of computations. The descriptions of the computations are obtained by "unrolling" the sequences of instructions into execution sequences. In the case of deterministic programs, each execution sequence provides a description for a computation. On the other hand, as it will be seen below, in the case of nondeterministic programs some execution sequences might be considered as computations, whereas others might be considered noncomputations. To delineate this distinction we need the following definitions.

An execution sequence is said to be an *accepting computation* if it terminates due to an accept command. An execution sequence is said to be a *nonaccepting computation* or a *rejecting computation* if it is on input that has no accepting computations. An execution sequence is said to be a *computation* if it is an accepting computation or a nonaccepting computation.

A computation is said to be a *halting computation* if it is finite.

Example 1.3.4 Consider the program in Figure 1.3.3.

```
read value
do
    write value
    value := value - 2
until value = 0
if eof then accept
```

Figure 1.3.3 A deterministic program.

Assume that the domain of the variables is the set of integers, with 0 as initial value.

On an input that consists of a single, even, positive integer, the program has an execution sequence that is an accepting computation (e.g., on input "4").

On an input that consists of more than one value and that starts with an even positive integer, the program has a halting execution sequence that is a nonaccepting computation (e.g., on input "4, 3, 2").

On the rest of the inputs the program has nonhalting execution sequences that are nonaccepting computations (e.g., on input "1"). ☐

An input is said to be *accepted* , or *recognized* , by a program if the program has an accepting computation on such an input. Otherwise the input is said to be *not accepted* , or *rejected* , by the program.

A program is said to have an *output* y on input x if it has an accepting computation on x with output y. The outputs of the nonaccepting computations are considered to be undefined , even though such computations may execute write instructions.

Example 1.3.5 The program in Example 1.3.4 (see Figure 1.3.3) accepts the inputs "2", "4", "6", . . . On input "6" the program has the output "6, 4, 2", and on input "2" the program has the output "2".

The program does not accept the inputs "0", "1", and "4, 2". For these inputs the program has no output, that is, the output is undefined. □

A computation is said to be a *nondeterministic computation* if it involves the execution of a nondeterministic instruction. Otherwise the computation is said to be a *deterministic computation*.

Nondeterministic Programs

Different objectives create the need for nondeterministic instructions in programming languages. One of the objectives is to allow the programs to deal with problems that may have more than one solution. In such a case, nondeterministic instructions provide a natural method of selection (see, e.g., Example 1.3.6 below). Another objective is to simplify the task of programming (see, e.g., Example 1.3.9 below). Still another objective is to provide tools for identifying difficult problems (see Chapter 5) and for studying restricted classes of programs (see Chapter 2 and Chapter 3).

Implementation considerations should not bother the reader at this point. After all, one usually learns the semantics of new programming languages before learning, if one ever does, the implementation of such languages. Later on it will be shown how a nondeterministic program can be translated into a deterministic program that computes a related function (see Section 4.3).

Nondeterministic instructions are essentially instructions that can choose between some given options. Although one is often required to make choices in everyday life, the use of such instructions might seem strange within the context of programs.

The semantics of a nondeterministic looping instruction of the form **do** α_1 **or** α_2 **or . . . or** α_k **until** Q(x_1, . . . , x_m), are similar to those of a deterministic looping instruction of the form **do** α **until** Q(x_1, . . . , x_m). The only difference is that in the deterministic case a fixed code segment α is executed in each iteration, whereas in the nondeterministic case an arbitrary code segment from α_1, . . . , α_k is executed in each iteration. The choice of a code segment can differ from one iteration to another.

Example 1.3.6 The program in Figure 1.3.4

```
counter := 0
/* Choose five input values. */
do
    read value
or
    read value
    write value
    counter := counter + 1
until counter = 5
/* Read the remainder of the input. */
do
    if eof then accept
    read value
until false
```

Figure 1.3.4 A nondeterministic program that chooses five input values.

is nondeterministic. The set of natural numbers is assumed to be the domain of the variables, with 0 as initial value. Parenthetical remarks are enclosed between /* and */.

The program on input "1, 2, 3, 4, 5, 6" has an execution sequence of the following form. The execution sequence starts with an iteration of the nondeterministic looping instruction in which the first code segment is chosen. The execution of the code segment consists of reading the input value 1, while writing nothing and leaving `counter` with the value of 0. Then the execution sequence continues with five additional iterations of the nondeterministic looping instruction. In each of the additional iterations, the second code segment is chosen. Each execution of the second code segment reads an input value, outputs the value that has been read, and increases the value of `counter` by 1. When `counter` reaches the value of 5, the execution sequence exits the first looping instruction. During the first iteration of the second looping instruction, the execution sequence halts due to the execution of the conditional accept instruction. The execution sequence is an accepting computation with output "2, 3, 4, 5, 6".

The program on input "1, 2, 3, 4, 5, 6" has four additional execution sequences similar to the one above. The only difference is that the additional execution sequences, instead of ignoring the input value 1, ignore the input values 2, 3, 4, and 5, respectively. An execution sequence ignores an input value i by choosing to read the value in the first code segment of the nondeterministic looping instruction. The additional execution sequences are accepting computations with outputs "1, 3, 4, 5, 6", "1, 2, 4, 5, 6", "1, 2, 3, 5, 6", and "1, 2, 3, 4, 6", respectively.

The program on input "1, 2, 3, 4, 5, 6" also has an accepting computation of the following form. The computation starts with five iterations of the first looping instruction. In each of these iterations the second code segment of the nondeterministic looping instruction is executed. During each iteration an input value is read, that value is written into the output, and the value of `counter` is increased by 1. After five iterations of the nondeterministic looping instruction, `counter` reaches the value of 5, and the computation transfers to the deterministic looping instruction. The computation reads the input value 6 during the first iteration of the deterministic looping instruction, and terminates during the second iteration. The output of the computation is "1, 2, 3, 4, 5".

The program has 2^7 - 14 execution sequences on input "1, 2, 3, 4, 5, 6" that are not computations. 2^6 - 7 of these execution sequences terminate due to trying to read beyond the input end by the first read instruction, and 2^6 - 7 of these execution sequences terminate due to trying to read beyond the input end by the second read instruction. In each of these execution sequences at least two input values are ignored by consuming the values in the first code segment of the nondeterministic looping instruction. The execution sequences differ in the input values they choose to ignore.

None of the execution sequences of the program on input "1, 2, 3, 4, 5, 6" is a nonaccepting computation, because the program has an accepting computation on such an input.

The program does not accept the input "1, 2, 3, 4". On such an input the program has 2^5 execution sequences all of which are nonaccepting computations.

The first nondeterministic looping instruction of the program is used for choosing the output values from the inputs. Upon choosing five values the execution sequences continue to consume the rest of the inputs in the second deterministic looping instruction.

On inputs with fewer than five values the execution sequences terminate in the first nondeterministic looping instruction, upon trying to read beyond the end of the inputs.

The variable `counter` records the number of values chosen at steps during each execution sequence. □

A deterministic program has exactly one execution sequence on each input, and each execution sequence of a deterministic program is a computation. On the other hand, the last example shows that a nondeterministic program might have more than one execution sequence on a given input, and that some of the execution sequences might not be computations of the program.

Nondeterministic looping instructions have been introduced to allow selections between code segments. The motivation for introducing nondeterministic assignment instructions is to allow selections between values. Specifically, a nondeterministic assignment instruction of the form x := ? assigns to the variable x an arbitrary value from the domain of the variables. The choice of the assigned value can differ from one encounter of the instruction to another.

Example 1.3.7 The program in Figure 1.3.5

```
/* Nondeterministically find a value that
 a. appears exactly once in the input, and
 b. is the last value in the input.                       */
last := ?
write last
/* Read the input values, until a value
equal to the one stored in last is reached.
*/
do
    read value
until value = last
/* Check for end of input. */
if eof then accept
reject
```

Figure 1.3.5 A nondeterministic program for determining a single appearance of the last input value.

is nondeterministic. The set of natural numbers is assumed to be the domain of the variables. The initial value is assumed to be 0.

The program accepts a given input if and only if the last value in the input does not appear elsewhere in the input. Such a value is also the output of an accepting computation. For instance, on input "1, 2, 3" the program has the output "3". On the other hand, on input "1, 2, 1" no output is defined since the program does not accept the input.

On each input the program has infinitely many execution sequences. Each execution sequence corresponds to an assignment of a different value to `last` from the domain of the variables.

An assignment to `last` of a value that appears in the input, causes an execution sequence to exit the looping instruction upon reaching such a value in the input. With such an assignment, one of the following cases holds.

a. The execution sequence is an accepting computation if the value assigned to `last` appears only at the end of the input (e.g., an assignment of 3 to `last` on input "1, 2, 3").
b. The execution sequence is a nonaccepting computation if the value at the end of the input appears more than once in the input (e.g., an assignment of 1 or 2 to `last` on input "1, 2, 1").
c. The execution sequence is not a computation if neither (a) nor (b) hold (e.g., an assignment of 1 or 2 to `last` on input "1, 2, 3").

An assignment to `last` of a value that does not appear in the input causes an execution sequence to terminate within the looping instruction upon trying to read beyond the end of the input. With such an assignment, one of the following cases hold.

a. The execution sequence is a nonaccepting computation if the value at the end of the input appears more than once in the input (e.g., an assignment to `last` of any natural number that differs from 1 and 2 on input "1, 2, 1").
b. The execution sequence is a nonaccepting computation if the input is empty (e.g., an assignment of any natural number to `last` on input " ").
c. The execution sequence is not a computation, if neither (a) nor (b) hold (e.g., an assignment to `last` of any natural number that differs from 1, 2, and 3 on input "1, 2, 3"). ☐

Intuitively, each program on each input defines "good" execution sequences, and "bad" execution sequences. The good execution sequences terminate due to the accept commands, and the bad execution sequences do not terminate due to accept commands. The best execution sequences for a given input are the computations that the program has on the input. If there exist good execution sequences, then the set of computations is identified with that set. Otherwise, the set of computations is identified with the set of bad execution sequences.

The computations of a program on a given input are either all accepting computations or all nonaccepting computations. Moreover, some of the nonaccepting computations may never halt. On inputs that are accepted the program might have execution sequences that are not computations. On the other hand, on inputs that are not accepted all the execution sequences are computations.

Guessing in Programs

The semantics of each program are characterized by the computations of the program. In the case of deterministic programs the semantics of a given program are directly related to the semantics of its instructions. That is, each execution of the instructions keeps the program within the course of a computation.

In the case of nondeterministic programs a distinction is made between execution sequences and computations, and so the semantics of a given program are related only in a restricted manner to the semantics of its instructions. That is, although each computation of the program can be achieved by executing the instructions, some of the execution sequences do not correspond to any computation of the program. The source for this phenomenon is the ability of the nondeterministic instructions to make arbitrary choices.

Each program can be viewed as having an imaginary agent with magical power that executes the program. On a given input, the task of the imaginary agent is to follow any of the computations the program has on the input. The case of deterministic programs can be considered as a lesser and restricted example in which the agent is left with no freedom. That is, the outcome of the execution of each deterministic instruction is completely determined for the agent by the semantics of the instruction. On the other hand, when executing a nondeterministic instruction the agent must satisfy not only the local semantics of the instruction, but also the global goal of reaching an accept command whenever the global goal is achievable.

Specifically, the local semantics of a nondeterministic looping instruction of the form **do** α_1 **or ... or** α_k **until** Q($x_1, \ldots, x_m$) require that in each iteration exactly one of the code segments $\alpha_1, \ldots, \alpha_k$ will be chosen in an arbitrary fashion by the agent. The global semantics of a program require that the choice be made for a code segment which can lead the execution sequence to halt due to a conditional accept instruction, whenever such is possible.

Similarly, the local semantics of a nondeterministic assignment instruction of the form x := ? require that each assigned value of x be chosen by the agent in an arbitrary fashion from the domain of the variables. The global semantics of the program require that the choice be made for a value that halts the execution sequence due to a conditional accept instruction, whenever such is possible.

From the discussion above it follows that the approach of "first guess a solution and then check for its validity" can be used when writing a program. This approach simplifies the task of the programmer whenever checking for the validity of a solution is simpler than the derivation of the solution. In such a case, the burden of determining a correct "guess" is forced on the agent performing the computations.

It should be emphasized that from the point of view of the agent, a guess is correct if and only if it leads an execution sequence along a computation of the program. The agent knows nothing about the problem that the program intends to solve. The only thing that drives the agent is the objective of reaching the execution of a conditional accept instruction at the end of the input. Consequently, it is still up to the programmer to fully specify the constraints that must be satisfied by the correct guesses.

Example 1.3.8 The program of Figure 1.3.6

```
/* Guess the output value. */
x := ?
write x
/* Check for the correctness of the
guessed value.                    */
do
    if eof then accept
    read y
until y = x
```

Figure 1.3.6 A nondeterministic program that outputs a noninput value.

outputs a value that does not appear in the input. The program starts each computation by guessing a value and storing it in x. Then the program reads the input and checks that each of the input values differs from the value stored in x. □

The notion of an imaginary agent provides an appealing approach for explaining nondeterminism. Nevertheless, the notion should be used with caution to avoid misconceptions. In particular, an imaginary agent should be employed only on full programs. The definitions leave no room for one imaginary agent to be employed by other agents. For instance, an imaginary agent that is given the program P in the following example cannot be employed by other agents to derive the acceptance of exactly those inputs that the agent rejects.

Example 1.3.9 Consider the program P in Figure 1.3.7. On a given input, P outputs an arbitrary choice of input values, whose sum equals the sum of the nonchosen input values. The values have the same relative ordering in the output as in the input.

```
sum1 := 0
sum2 := 0
do
    if eof then accept
    do                  /* Guess where the next input value belongs. */
        read x
        sum1 := sum1 + x
    or
        read x
        write x
        sum2 := sum2 + x
    until sum1 = sum2         /* Check for the correctness of the
               guesses, with respect to the portion
               of the input consumed so far.
            */
```

```
until false
```

Figure 1.3.7 A nondeterministic program for partitioning the input into two subsets of equal sums of elements.

For instance, on input "2, 1, 3, 4, 2" the possible outputs are "2, 1, 3", "1, 3, 2", "2, 4", and "4, 2". On the other hand, no output is defined for input "2, 3".

In each iteration of the nested looping instruction the program guesses whether the next input value is to be among the chosen ones. If it is to be chosen then `sum2` is increased by the magnitude of the input value. Otherwise, `sum1` is increased by the magnitude of the input value. The program checks that the sum of the nonchosen input values equals the sum of the chosen input values by comparing the value in `sum1` with the value in `sum2`. □

Example 1.3.10 The program of Figure 1.3.8 outputs the median of its input values, that is, the $\lceil n/2\rceil$ th smallest input value for the case that the input consists of n values. On input "1, 3, 2" the program has the output "2", and on input "2, 1, 3, 3" the program has the output "3".

```
median := ?          /* Guess the median. */
write median
count := 0
do
    /* Find the difference between the
    number of values greater than and those
    smaller than the guessed median.
 */
    do
        read x
        if x > median then
            count := count + 1
      if x < median then
            count := count - 1
      if x = median then
            do
                count := count + 1
         or
                count := count - 1
         until true
   until 0 ≤ count ≤ 1
   /* The median is correct for the portion of the
    input consumed so far.                          */
    if eof then accept
until false
```

Figure 1.3.8 A nondeterministic program that finds the median of the input values.

The program starts each computation by storing in `median` a guess for the value of the median. Then the program reads the input values and determines in `count` the difference between the number of input values that are greater than the one stored in `median` and the number of input values that are smaller than the one stored in `median`.

For those input values that are equal to the value stored in `median`, the program guesses whether they should be considered as bigger values or smaller values.

The program checks that the guesses are correct by verifying that `count` holds either the value 0 or the value 1. □

The *relation computed* by a program P, denoted R(P), is the set { (x, y) | P has an accepting computation on input x with output y }. When P is a deterministic program, the relation R(P) is a function.

Example 1.3.11 Consider the program P in Figure 1.3.6. Assume the set of natural numbers for the domain of the variables. The relation R(P) that P computes is { (α, a) | α is a sequence of natural numbers, and a is a natural number that does not appear in α }. □

The language that a program P accepts is denoted by L(P) and it consists of all the inputs that P accepts.

Configurations of Programs

An execution of a program on a given input is a discrete process in which the input is consumed, an output is generated, the variables change their values, and the program traverses its instructions. Each stage in the process depends on the outcome of the previous stage, but not on the history of the stages. The outcome of each stage is a configuration of the program that indicates the instruction being reached, the values stored in the variables, the portion of the input left to be read, and the output that has been generated so far. Consequently, the process can be described by a sequence of moves between configurations of the program.

Formally, a segment of a program is said to be an *instruction segment* if it is of any of the following forms.

a. Read instruction
b. Write instruction
c. Assignment instruction
d. **if** Q(x_1, . . . , x_m) **then** portion of a conditional if instruction
e. **do** portion of a looping instruction
f. **until** Q(x_1, . . . , x_m) portion of a looping instruction
g. Conditional accept instruction
h. Reject instruction

Consider a program P that has k instruction segments, m variables, and a domain of variables that is denoted by D. A *configuration* , or *instantaneous description*, of P is a five-tuple (i, x, u, v, w), where $1 \leq i \leq k$, x is a sequence of m values from D, and u, v, and w are sequences of values from D.

Intuitively, a configuration (i, x, u, v, w) says that P is in its *i*th instruction segment, its *j*th variable contains the *j*th value of x, u is the portion of the input that has already been read, the leftover of the input is v, and the output so far is w. (The component u is not needed in the definition of a configuration. It is inserted here for reasons of compatibility with future definitions that require such a component.)

Example 1.3.12 Consider the program in Figure 1.3.9.

```
last := ?                    /* I_1 */
write last                          /* I_2 */
do                                     /* I_3 */
    read value                     /* I_4 */
until value = last              /* I_5 */
if eof then accept             /* I_6 */
```

```
        reject                                    /* I7 */
```

Figure 1.3.9 A program consisting of seven instruction segments.

Assume the set of natural numbers for the domain D of the variables, with 0 as initial value. Each line I_i in the program is an instruction segment. The program has k = 7 instruction segments, and m = 2 variables.

In each configuration (i, x, u, v, w) of the program i is a natural number between 1 and 7, and x is a pair <last, value> of natural numbers that corresponds to a possible assignment of last and value in the variables `last` and `value`, respectively. Similarly, u, v, and w are sequences of natural numbers.

The configuration (1, <0, 0>, <>, <1, 2, 3>, <>) states that the program is in the first instruction segment, the variables hold the value 0, no input value has been read so far, the rest of the input is "1, 2, 3", and the output is empty.

The configuration (5, <3, 2>, <1, 2>, <3>, <3>) states that the program is in the fifth instruction segment, the variable `last` holds the value 3, the variable `value` holds the value 2, "1, 2" is the portion of the input consumed so far, the rest of the input contains just the value 3, and the output so far contains only the value 3. □

A configuration (i, x, u, v, w) of P is called an *initial configuration* if i = 1, x is a sequence of m initial values, u is an empty sequence, and w is an empty sequence. The configuration is said to be an *accepting configuration* if the *i*th instruction segment of P is a conditional accept instruction and v is an empty sequence.

A direct move of P from configuration C_1 to configuration C_2 is denoted $C_1 \vdash_P C_2$, or simply $C_1 \vdash C_2$ if P is understood. A sequence of unspecified number of moves of P from configuration C_1 to configuration C_2 is denoted $C_1 \vdash_P {}^* C_2$, or simply $C_1 \vdash^* C_2$ if P is understood.

Example 1.3.13 Consider the program in Figure 1.3.9. On input "1, 2, 3" it has an accepting computation that goes through the following sequence of moves between configurations. The first configuration in the sequence is the initial configuration of the program on input "1, 2, 3", and the last configuration in the sequence is an accepting configuration of the program. In each configuration (i, x, u, v, w) the pair x = <last, value> corresponds to the assignment of last and value in the variables `last` and `value`, respectively.

$$\begin{array}{rl}
(1,\langle 0,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle\rangle) & \vdash (2,\langle 3,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle\rangle) \\
& \vdash (3,\langle 3,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle 3\rangle) \\
& \vdash (4,\langle 3,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle 3\rangle) \\
& \vdash (5,\langle 3,1\rangle,\langle 1\rangle,\langle 2,3\rangle,\langle 3\rangle) \\
& \vdash (3,\langle 3,1\rangle,\langle 1\rangle,\langle 2,3\rangle,\langle 3\rangle) \\
& \vdash (4,\langle 3,1\rangle,\langle 1\rangle,\langle 2,3\rangle,\langle 3\rangle) \\
& \vdash (5,\langle 3,2\rangle,\langle 1,2\rangle,\langle 3\rangle,\langle 3\rangle) \\
& \vdash (3,\langle 3,2\rangle,\langle 1,2\rangle,\langle 3\rangle,\langle 3\rangle) \\
& \vdash (4,\langle 3,2\rangle,\langle 1,2\rangle,\langle 3\rangle,\langle 3\rangle) \\
& \vdash (5,\langle 3,3\rangle,\langle 1,2,3\rangle,\langle\rangle,\langle 3\rangle) \\
& \vdash (6,\langle 3,3\rangle,\langle 1,2,3\rangle,\langle\rangle,\langle 3\rangle)
\end{array}$$

The subcomputation

$$(1,\langle 0,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle\rangle) \vdash^* (1,\langle 0,0\rangle,\langle\rangle,\langle 1,2,3\rangle,\langle\rangle)$$

consists of zero moves, and the subcomputation

$$(1, <\bar{0}, \bar{0}>, <>, <1, 2, 3>, <>) \vdash^* (6, <3, 3>, <1, 2, 3>, <>, <3>)$$

consists of eleven moves. □

[next] [prev] [prev-tail] [front] [up]