Computability in an Introductory Course on Programming

Hans J. Schneider
Lehrstuhl für Programmiersprachen, Universität Erlangen-Nürnberg,
Martensstraße 3, D-91058 Erlangen
Electronic Mail: schneider@informatik.uni-erlangen.de

Abstract:

The programming approach to computability presented in the textbook by Kfoury, Moll, and Arbib in 1982 has been embedded into a programming course following the textbook by Abelson and Sussman. This leads to a course concept teaching good programming practice and clear theoretical concepts simultaneously. Here, we explain some of the main points of this approach: the halting problem, primitive and μ-recursive functions and the operational counterpart of these functions, i.e., the Loop and the While programs.

1 Introduction

A usual programming course is mainly concerned with presenting syntactic sugar and semantic tricks to solve more or less small, standardized problems. In the end, the students know a lot of details how to make a computer run, and at the best, they are familiar with good design practice and some software-engineering tools. But, do they understand what computer science is and what it can bring about? To remedy this, they need not only be taught the theory as a supplement, but they should be supplied with the relationship between fundamentals and practice.

The main task of both programming and theory is making definitions, proving properties of the objects introduced by the definitions, and taking advantage of these properties. We can derive profit from this analogy by teaching theory from a programmer's point of view. A topic, which is very closely related to programming and at the same time, is at the heart of theoretical computer science, is computability theory. Therefore, it is well-suited as an entrance to theory. Even freshmen perceive the clarity of Kleene's theory of recursive functions if the definitions are translated one-to-one into higher-order functions of about five lines using some LISP-dialect. The students can then apply these functions to create computable functions one after the other, make them run and can convince themselves that the theory works. It is nice to see that even hackers can be tempted into studying the theory of recursive functions, efficiency problems, denotational semantics, etc., if they can do this sitting in front of their beloved computers.

In 1995, Morales-Bueno has proposed to explain noncomputability in a first-year programming course with the aid of a programming language version of the busy-beaver-problem [7]. He supposes that the halting problem is cited in this context for historical reasons and he argues that a formal proof of the unsolvability of the halting problem is beyond the scope of such a course. From his paper and some discussions we had with colleagues, we have learned that a very interesting and easy-to-understand approach to unsolvability of the halting problem is no longer known in the community and that it should be brought to the mind of a broader audience again. We think of the approach already presented by Hoare and Allison in 1972 [3].

In the following, we summarize two of the theoretical aspects we have integrated into our introductory course. First, we present our variation of the Hoare-Allison argument; then, we

consider Kleene's approach to computable functions mainly following the textbook of Kfoury et al. [4]. Our introductory course is based on the textbook of Abelson and Sussman [1] using the functional programming language SCHEME. The concept is now used since more than ten years.

2 Unsolvability of the Halting Problem

SCHEME is a functional programming language. Therefore, the first control construct to be used by the students is recursion and with high probability, they encounter the problem of nonterminating programs very early. They are motivated to ask for a program testing their own programs for termination. At this point, we prove:

Theorem 1 (Halting problem):
In a type-free language containing alternatives, negation, and recursive functions, it is impossible to write a program that solves the halting problem.

The proof follows Hoare and Allison and is by contradiction[1]. We assume existence of a SCHEME program solving the halting problem:

Definition 2 (Type-free terminates function):
The program

```
(define (terminates? p x) ( body ))
```

solves the halting problem if the body satisfies the following properties:

(a) The function terminates for all values of the parameters `p` and `x`.
(b) The function call yields `true` if application of `p` to `x`, i.e., the prospective call `(p x)`, terminates.
(c) It yields `false` if `(p x)` does not terminate.

Under this assumption, we can write another SCHEME function:

Program 3 (Hetero):

```
(define (hetero p)
        (cond ((terminates? p p) (not (p p)))
              (else #t)
)       )
```

This function always terminates since `(p p)` is called only after ensuring its termination. (In the other branch, the constant `true` is returned; this is trivially a terminating function.) Since each call to `hetero` terminates, the calls `(terminates? hetero p)` yield true for all `p`. Therefore, `(terminates? hetero hetero)` also yields `true`, and this tells us that `(hetero hetero)` terminates. If, however, we want to determine the result of this call, we get a contradiction

```
(hetero hetero) = not (hetero hetero)
```

and therefore, our assumption is refuted. The function `hetero` is closely related to Russel's paradox [3]: An adjective is said to be heterological if it does not apply to itself, and we may ask the question whether "heterological" is heterological.

[1] A similar proof has been presented by Kfoury et al. [4, p. 11]. The idea seems to originate from Strachey [9].

Attentative students may now argue that we have used an unfair trick. In a safe programming language, type-checking prevents `p` to be applied to itself. We can, however, construct an intuitive variation of the argument by writing two functions using a language with static typing. The idea is to use the data type `string` to describe functions as well as all the arguments, as we do when inputting them into the computer. The following notation uses PASCAL syntax, but may easily be replaced by other languages, e.g., a functional language with type checking as ML or MIRANDA:

Definition 4 (Typed terminates function):
We assume existence of two programs:

```
function interpret(p, x: string) boolean:
begin
      body1
end;
function terminates(p, x: string) boolean:
begin
      body2
end
```

the bodies of which satisfy the following properties:

(a) If `p` is not a syntactically correct PASCAL program, `body1` does not terminate, i.e., the interpreter enters an infinite loop (instead of printing an error message).
(b) If `p` is a syntactically correct PASCAL program, `body1` interprets `p` with data `x`.
(c) `body2` terminates for all arguments `p` and `x`; it returns `true` if and only if the function call `interpret(p, x)` terminates, i.e., if `p` is a syntactically correct PASCAL program and if its interpretation with data `x` terminates.

Then, we rewrite the function `hetero` in the following way:

Program 5:

```
function hetero(p: string) boolean:
begin
      if terminates(p, p) then not interpret(p, p)
                          else true
end
```

The call `interpret(p, p)` terminates if `p` is syntactically correct and running the program `p` with the data `p` terminates. Again, we get the contradiction:

```
interpret(hetero, hetero) = not interpret(hetero, hetero)
```

But in this case, we have assumed two function bodies to exist. Therefore, we can conclude only the following:

Theorem 6 (Halting problem in typed languages):
In a statically typed programming language containing the data types of strings and truth values, alternatives, negation, and function calls, we have at most one of the following possibilities:

- We can write the interpreter of the language in the language itself.
- We can solve the halting problem for this language in the language itself.

Since existence of a compiler implies existence of an interpreter[2] and since there are Pascal-compilers written in Pascal, the halting problem can not be solved in Pascal. Conversely, if you define a programming language allowing only terminating programs, you can not write the compiler or the interpreter in this language, since the halting problem is solved by a trivial body returning `true` in each case.

3 Primitive Recursive Functions

Using only one data type makes a fine opportunity to introduce the notion of goedelization, since each string can be interpreted as a (binary) natural number [4, p. 47]. Of course, Goedel's original approach uses prime-number factorization, but the arguments are very analogous: Using the prime number approach, we also have to check whether the decomposition of a number fulfills a certain formation rule (syntactic correctness), and we assign an interpretation only to the correct numbers. An introductory course is not the right place to go into the details of goedelization, but the idea can be used to explain why it is sufficient to restrict computability considerations to functions of natural numbers, and we can discuss the limits of computability along Kleene's theory of recursive functions.

Definition 7 (Primitive recursive functions):
Primitive recursive functions are defined inductively:

(a) The constant function $c_0 = 0$, the successor function $\mathrm{succ}(x) = x+1$, and the projections $p_{ni}(x_1, \ldots, x_n) = x_i$ are primitive recursive.
(b) The class of primitive recursive functions is closed under substitution.
(c) If $g : \mathbb{N}^n \to \mathbb{N}$ and $h : \mathbb{N}^{n+2} \to \mathbb{N}$ are primitive recursive, then the function $f : \mathbb{N}^{n+1} \to \mathbb{N}$ defined by

$$\begin{array}{lllll} f(c_0, & x_1, \ldots, x_n) & = & g & (\ x_1, \ldots, x_n\ ) \\ f(succ(y), & x_1, \ldots, x_n) & = & h & (\ f(y, x_1, \ldots, x_n), \\ & & & & \quad y, \\ & & & & \quad x_1, \ldots, x_n \\ & & & &) \end{array}$$

is primitive recursive, too.

The basic functions of this definition can easily be translated into SCHEME:

Program 8 (Basic functions):

```
(define c0 0)

(define (succ x) (+ x 1))

(define (p11 x) x)

(define (p21 x y) x)
(define (p22 x y) y)
```

[2] Run the compiler and subsequently run the object program: The effect is the same as running an interpreter.

```
(define (p31 x y z) x)
(define (p32 x y z) y)
(define (p33 x y z) z)
...
```

Substitution is available in all programming languages. The only problem we encounter is the fact that the left-hand side of the definition of primitive recursion includes $\mathrm{succ}(y)$.[3] Therefore, we add the predecessor function $\mathrm{pred}(x) = x - 1$ to the set of basic functions:

Program 9 (Predecessor):

```
(define (pred x) (- x 1))
```

This does not change the class of primitive recursive functions since the predecessor function can be defined by primitive recursion with $g = c_0$ and $h = p_{21}$.

Now, we are ready for implementing primitive recursion. For simplicity, we define different higher-order functions depending on the number of arguments of the function to be constructed[4]. `prim-rec-0` constructs functions with one argument, i.e., in the definition, we have $n = 0$, `prim-rec-1` corresponds to $n = 1$, etc.:

Program 10 (Primitive Recursion):

```
(define (prim-rec-0 g h)
    (define (f y)
        (cond ((= y c0) g)
              (else (h (f (pred y)) (pred y)))
    )   )
    f
)

(define (prim-rec-1 g h)
    (define (f y x)
        (cond ((= y c0) (g x))
              (else (h (f (pred y) x) (pred y) x))
    )   )
    f
)
```

We give examples of constructing primitive recursive functions by applying these higher-order functions:

[3] This is no problem if we use a modern functional language providing pattern matching.

[4] At this point of our course, we have not yet introduced SCHEME functions which allow an arbitrary number of arguments.

Program 11 (Examples):

```
(define odd (prim-rec-0 c0 (lambda (z y)
                                (sub (succ c0) z)
)           )              )
(define add
        (prim-rec-1 p11
                    (lambda (z y x)
                            (succ (p31 z y x))
)       )           )

(define subi
        (prim-rec-1 p11
                    (lambda (z y x)
                            (pred (p31 z y x))
)       )           )

(define sub
        (lambda (x y)
                (subi (p22 x y) (p21 x y))
)       )
```

Here `sub` denotes the modified subtraction $sub(y, x) = y \dot{-} x$ and `subi` its inverse function $subi(y, x) = x \dot{-} y$. The students are asked to test some of the more complicated examples given in a usual textbook, e.g., in [4]. These examples include primitive recursive predicates and the bounded μ-operator.

4 Loop programs

This may lead to ask for "traditional" language constructs that allow to implement the primitive recursive functions and nothing else, i.e., to consider the theoretical foundations of the operational programming paradigm. For this purpose, Meyer and Ritchie [6] have introduced the notion of Loop programs, and they have shown that they are equivalent to the primitive recursive functions:

Definition 12 (Loop programs):
A Loop program is a finite sequence of instructions for changing non-negative integers stored in registers. The instructions are of four types:

(a) constant zero: $X = 0$
(b) increment: $X = X + 1$
(c) assignment: $X = Y$
(d) bounded loop: $LOOP\ X\ <$ sequence of instructions $>\ END$

where X and Y denote the names of registers.[5]

Abelson and Sussman [1, Section 5.1] present a simulator for register machines, but we prefer a simpler solution tailored for our purpose. We encode Loop programs as structured lists in

[5]The assignment operation does not increase the generative power of Loop programs, but makes programming more convenient.

SCHEME. Consider, e.g., the following programs to compute the predecessor function and the assignment, respectively:

Program 13:

```
(define loop-pred
        '((regs a b c)
          (zero b)
          (zero c)
          (loop a (copy c b)
                  (incr b)
          )
          (copy a c)
)        )
```

```
(define loop-assign
        '((regs a b)
          (zero a)
          (loop b (incr a)
          )
)         )
```

The result is always found in register `a`. When calling the interpreter, the given parameters are stored into the registers at the beginning of the `regs` list. The other registers are auxiliary registers. The example `loop-pred` may be called by `(interpret loop-pred '(4))`. Register `a` contains the parameter and at the end of the run, the result. Registers `b` and `c` are used to store intermediate values.

The interpreter uses recursive-descent parsing:

Program 14 (Interpreter for Loop programs):

```
(define (interpret progr data)
   (initialize-regs (car progr) data)
   (interpret-body (cdr progr))
   (get-reg 'a)
)

(define (initialize-regs stat data)
   (define (helper  rlist data)
      (cond ((null? rlist) nil)
            (else (cond ((null? data)
                         (def-reg! (car rlist) 0)
                         (helper (cdr rlist) data))
                        (else
                         (def-reg! (car rlist) (car data))
                         (helper (cdr rlist) (cdr data)))
   )   )    )     )
   (set! reglist '())
   (cond ((eq? (car stat) 'regs)
              (helper (cdr stat) data))
)  )

(define (interpret-body body)
   (cond ((null? body) nil)
         (else (let ((key (caar body))
```

```
                        (pars (cdar body))
                        (tail (cdr body))
                       )
                       (cond ((eq? key 'zero)  (interpret-zero pars))
                             ((eq? key 'incr)  (interpret-incr pars))
                             ((eq? key 'copy)  (interpret-copy pars))
                             ((eq? key 'loop)  (interpret-loop pars))
                             ((eq? key 'while) (interpret-while pars))
                       )
                       (interpret-body tail)
  )  )     )     )
```

The basic instructions of the Loop language can be implemented straightforwards:

Program 15 (Basic instructions):

```
(define (int-zero pars)
   (set-reg! (car pars) 0)
)

(define (int-incr pars)
   (incr-reg (car pars))
)

(define (int-copy pars)
   (set-reg! (car pars) (get-reg (cadr pars)))
)

(define (int-loop pars)
   (letrec ((reps (get-reg (car pars)))
            (body (cdr pars))
            (iter (lambda (rep)
               (cond ((equal? rep 0) nil)
                     (else (int-body body)
                           (iter (- rep 1))
            ))  )  )  )
            (iter reps)
)  )
```

For reason of space, we have omitted the syntax checks in the version presented here. Instead, we assume that the syntax of the instructions is correct, e.g., that the `copy` instruction has a parameter list consisting of the names of two registers declared in the prelude. This assumption also simplifies the procedures that interact with the list of registers:

Program 16 (Register administration):

```
(define reglist '())

(define (get-reg r)
```

```
   (cdr (assq r reglist))
)

(define (set-reg! r v)
   (set-cdr! (assq r reglist) v)
)

(define (incr-reg r)
   (let ((reg (assq r reglist)))
        (set-cdr! reg (+ 1 (cdr reg)))
)  )

(define (def-reg! reg val)
   (define (last reglist)
      (cond ((null? (cdr reglist)) reglist)
            (else (last (cdr reglist)))
   )  )
   (cond ((null? reglist) (set! reglist (list (cons reg val))))
         (else (set-cdr! (last reglist) (list (cons reg val))))

)  )
```

It is an instructive exercise to add syntax checks to our interpreter. The usual solution is adding suitable error messages. The interpreter matches the definition of Definition 4, if we substitute an infinite loop for the standard function `error`.

5 μ-Recursive Functions

It is very easy to show that there exist computable functions that are not primitive recursive [4, p. 208]. Although the proof is very simple, the students may ask for an example. In theoretical computer science, the usual answer to this question is the Ackermann function, but it is very difficult to prove that this function is not primitive recursive (see, e.g., [2, p. 82-88]), and furthermore, it is not a function relevant in the everday applications. Since primitive recursive functions are total, each Loop program terminates, and from Theorem 6, we get:

Corollary 17: An interpreter (or a compiler) for the language of Loop programs can not be written as a Loop program, i.e., it is not a primitive recursive function.

Interpreters (and compilers) not only are examples of practical relevance, but also do not show the rapid growth of the Ackermann function. To extend our computability concepts, we need language constructs that allow infinite loops.

Definition 18 (μ-recursive functions):
A function is μ-recursive if
- it is one of the basic primitive recursive functions,
- it is constructed from μ-recursive functions by substitution,
- it is constructed from μ-recursive functions by primitive recursion,

– it is constructed from a total μ-recursive function by application of the (unbounded) μ-operator:

$$\mu_y \; h(y, x_1, \ldots, x_n) = \begin{cases} y_0 & \text{if } h(y_0, x_1, \ldots, x_n) = c0 \\ & \quad \wedge (\forall y < y_0)(h(y, x_1, \ldots, x_n) \neq c0) \\ \text{undefined} & \text{if there is no such } y \end{cases}$$

We can directly translate μ-recursion into a SCHEME program. Here, we consider only the case that we want to define a function with two arguments, the other cases are analogous:

Program 19 (μ-recursion):

```
(define (mu-rek-2 h)
   (define (helper y x1 x2)
      (cond ((= (h y x1 x2) c0) y)
            (else (helper (succ y) x1 x2))
   )  )
   (lambda (x1 x2) (helper c0 x1 x2))
)
```

The local function `helper` implements the unbounded iteration; it is called with the argument $y = 0$ and then, it calls itself again after incrementing the argument y.

The operational analogue is the language of While programs:

Definition 20 (While programs):
A While program is a finite sequence of instructions for changing non-negative integers stored in registers. The instructions are of four types:

(a) constant zero: $X = 0$
(b) increment: $X = X + 1$
(c) assignment: $X = Y$
(d) unbounded loop: $WHILE\ X \neq Y\ DO\ <$ sequence of instructions $>\ END$

where X and Y denote the names of registers.

The interpreter of the While language can easily be derived from the interpreter of the Loop language, since it is sufficient to replace the Loop construct with an unbounded loop:

Program 21 (Interpreter for While programs):

```
(define (interpret-while pars)
   (define reg1 (car pars))
   (define reg2 (cadr pars))
   (define body (caddr pars))
   (define (helper)
      (cond ((equal? (get-reg reg1) (get-reg reg2)) nil)
            (else (interpret-body body)
                  (helper)  ; repeat loop
   )  )     )
   (helper)                 ; first call
)
```

A simple example is the implementation of integer division:

Program 22 (Integer division):

```
(define w-div
        '((regs a b c d e)
          (copy c a)
          (zero a) (zero d)
          (while d c ((zero e)
                      (while e b ((incr d)
                                  (incr e)
                      )          )
                      (incr a)
)        ))          )
```

This function yields a/b if the result is integer and enters an infinite loop, otherwise.

From the interpreter of While programs, the students get a first impression of what a universal function is. For this, we have to show that the interpreter can be written as a While program. Although this proof needs some more pages, it is straightforward. The students can implement and test the necessary subroutines along the exercises given in [4, Sect. 3.1].

6 Conclusion

When revising our course about ten years ago, we decided to follow the textbook by Abelson and Sussman [1], since a stable SCHEME interpreter was available for free, and every student could be assumed to install it on its own computer. Today, one may choose other functional languages, e.g., ML. MIRANDA would be even better because of some more convenient notations such as the `where`-clause instead of `let` and indentation having syntactical meaning. Unfortunately, MIRANDA is more expensive and - as far as we know - it is no longer supported.

We have introduced some additional theoretical aspects step by step into the course. Our presentation of computability results was strongly influenced by the fine textbook of Kfoury, Moll, and Arbib [4], although we have translated the programming language ideas into SCHEME. Since functional languages cause students to write recursive programs and since the fast increasing number of parentheses of SCHEME programs enforces subdividing programs into small pieces, it is easy to verify the correctness of each program. (Even if the students do not so later on, they have adopted a special style of thinking about program structure.) Other theoretical aspects can be introduced in a secondary course on functional programming, e.g., denotational semantics, and the concepts typical of compiler technique can be added to the interpreter in a straightforward way.

At many universities, teaching practical programming is separated from teaching computer science. As a result, students consider programming and computer science as two different worlds. We have not only to look for how to make the students enthusiastic about theory, but also to train them for applying rigorous thinking to programming, and this can be done best by teaching both things together.

References

[1] H. Abelson/G.J. Sussman: *Structure and Interpretation of Computer Programs*, MIT Press, 1985

[2] H. Hermes: *Enumerability, Decidability, Computability*, Springer, 1965 (2nd Edition), pp. 82-88

[3] C.A.R. Hoare/D.C.S. Allison: Incomputability, *Computing Surveys 4*, 3 (Sept. 1972), pp. 169-178

[4] A.J. Kfoury/R.N. Moll/M.A. Arbib: *A Programming Approach to Computability*, Springer, 1982

[5] S.C. Kleene: General recursive functions of natural numbers, *Mathematische Annalen* 112, 5 (1936), pp. 727-742

[6] A.R. Meyer/D.M. Ritchie: The complexity of loop programs, *Proceedings of the 22nd ACM National Conference* (1967), pp. 465-469

[7] R. Morales-Bueno: Noncomputability is easy to understand, *Commun. ACM 38*, 8 (Aug. 1995), pp. 116-117

[8] R. Morales-Bueno et al.: Two classical theorems revisited, *EATCS Bulletin* 71 (2000), pp. 204-215

[9] C. Strachey: An impossible program, Letter to the Editor, *The Computer J.* 8 (1965), p. 313